

FIG. 1

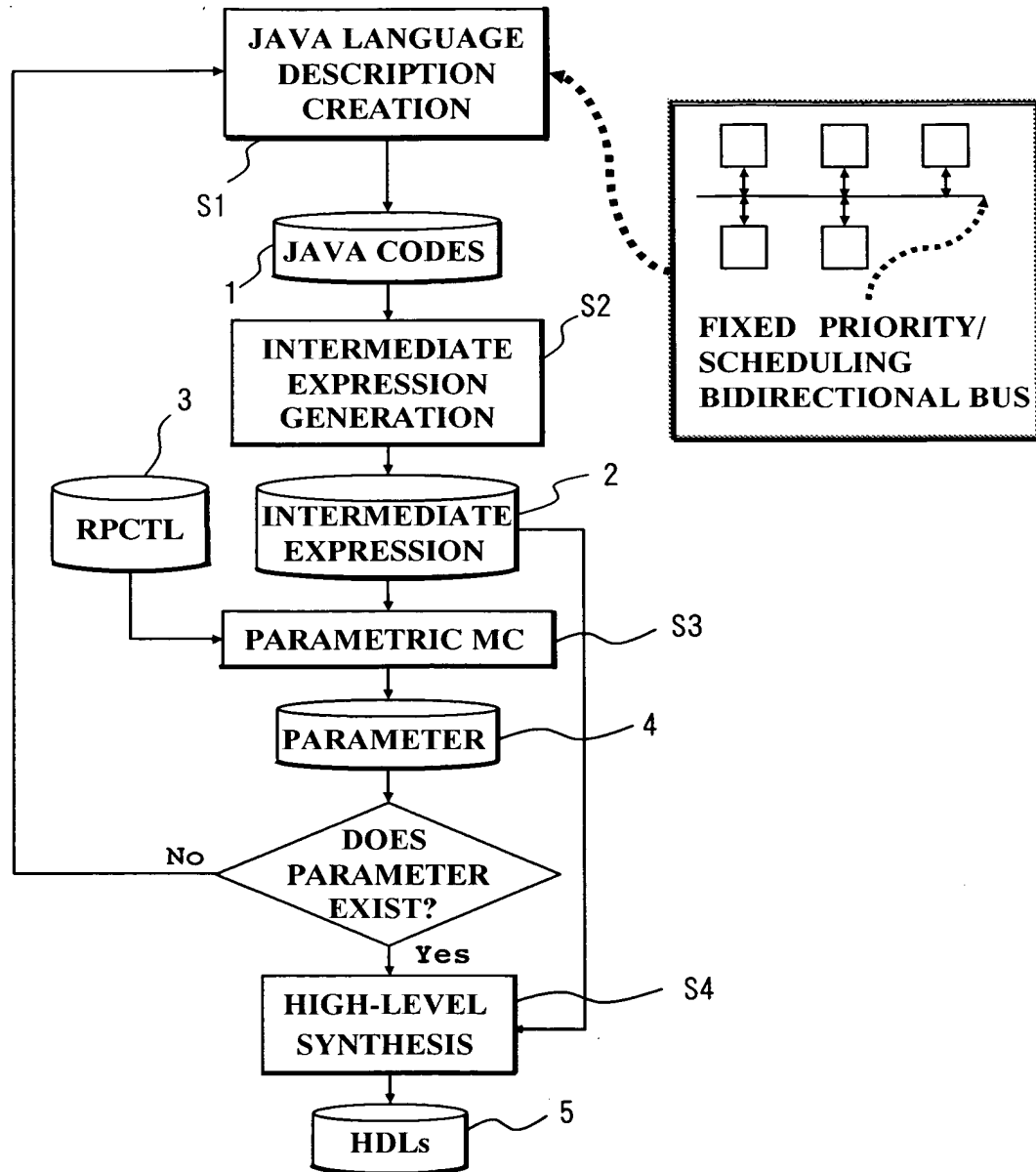


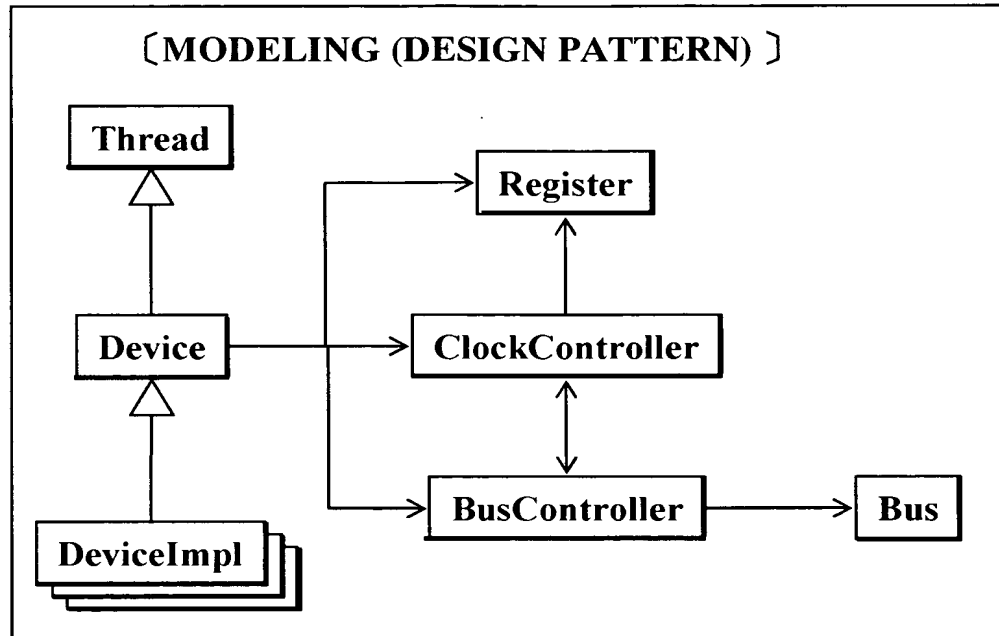
FIG. 2

FIG. 3**[MODELING (CLOCK SYNCHRONIZATION)]**

```

private void consume_1_clock() {
    /* CHECK IF NUMBER OF ALL DEVICES IS EQUAL TO NUMBER OF
       DEVICES HAVING EXECUTED THIS METHOD */
    if (++this.current_num == this.device_num) {
        this.current_num = 0;
        for (int i=0; i<this.device_num; i++) {
            /* EXECUTE REGISTER ASSIGNMENT */
            registers[i].assignWriteValue();
        }
        /* INITIALIZE FLAG VARIABLE FOR IDENTIFYING BUS LOCK
           (BUS ACCESS FLAG) */
        if (this.bc.getBusyCount() == 0) {
            this.bc.initLockDoneOnceFlag();
        }
        notifyAll();
    } else {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
}

```

FIG. 4**[MODELING (CLOCK SYNCHRONIZATION)]**

```

public void assignWriteValue() {
    /* DECIDE IF WRITING INTO REGISTER HAS BEEN EXECUTED BY
       "sync_write" METHOD OR "sync_burst_write" METHOD TO BE STATED
       LATER */
    if (this.write_access) {
        /* EXECUTE WRITING INTO REGISTER (ARRAY index)
           ACTUALLY SUBJECTED TO WRITING */
        this.current_value[this.update_index] = this.write_value;
        /* RESET RIGHT ACCESS FLAG */
        this.write_access = false;
    }
}

```

FIG.5

[MODELING (BUS OWNERSHIP ACQUISITION)]	
METHOD NAME	
getBusLock	BUS OWNERSHIP IS ACQUIRED BY CALLING "lock" METHOD WHEN "tryGetBusLock" RETURNS "true", AND IN CASE OF "false", "consume_clock" IS CALLED SO AS TO CONSUME CLOCKS IN NUMBER OF "clock_num".
tryGetBusLock	IN CASE WHERE "getBusOwner" IS "null" AND WHERE "LockBusDoneOnceFlag" IS "false", "lock" IS CALLED, AND "true" IS RETURNED TO "getBusLock". "false" IS RETURNED OTHERWISE.
getBusOwner	OBJECT OF "thread" CURRENTLY LOCKING BUS IS RETURNED. IF LOCKING BUS IS NONEXISTENT, "null" IS RETURNED.
lock	"Thread" WHICH IS CURRENTLY EXECUTING "Bus class" OBJECT ("lock" METHOD) IS REGISTERED AS OBJECT CURRENTLY LOCKING BUS.
getBusDoneOnceFlag	VALUE OF BUS ACCESS FLAG IS ACQUIRED.
consume_clock	"consume_1_clock" IS CALLED.
consume_1_clock	AS ALREADY EXPLAINED ON CLOCK SYNCHRONIZATION MECHANISM.
initLockDoneOnceFlag	"setLockBusDoneOnceFlag" IS CALLED BY SETTING ARGUMENT AT "false".
setLockBusDoneOnceFlag	ASSIGN ARGUMENT TO BUS ACCESS FLAG.
assignWriteValue	IN CASE WHERE RIGHT ACCESS TO SHARED REGISTER IS EXISTENT, REGISTER ASSIGNMENT IS EXECUTED BEFORE PROCEEDING TO NEXT CLOCK.

FIG. 6

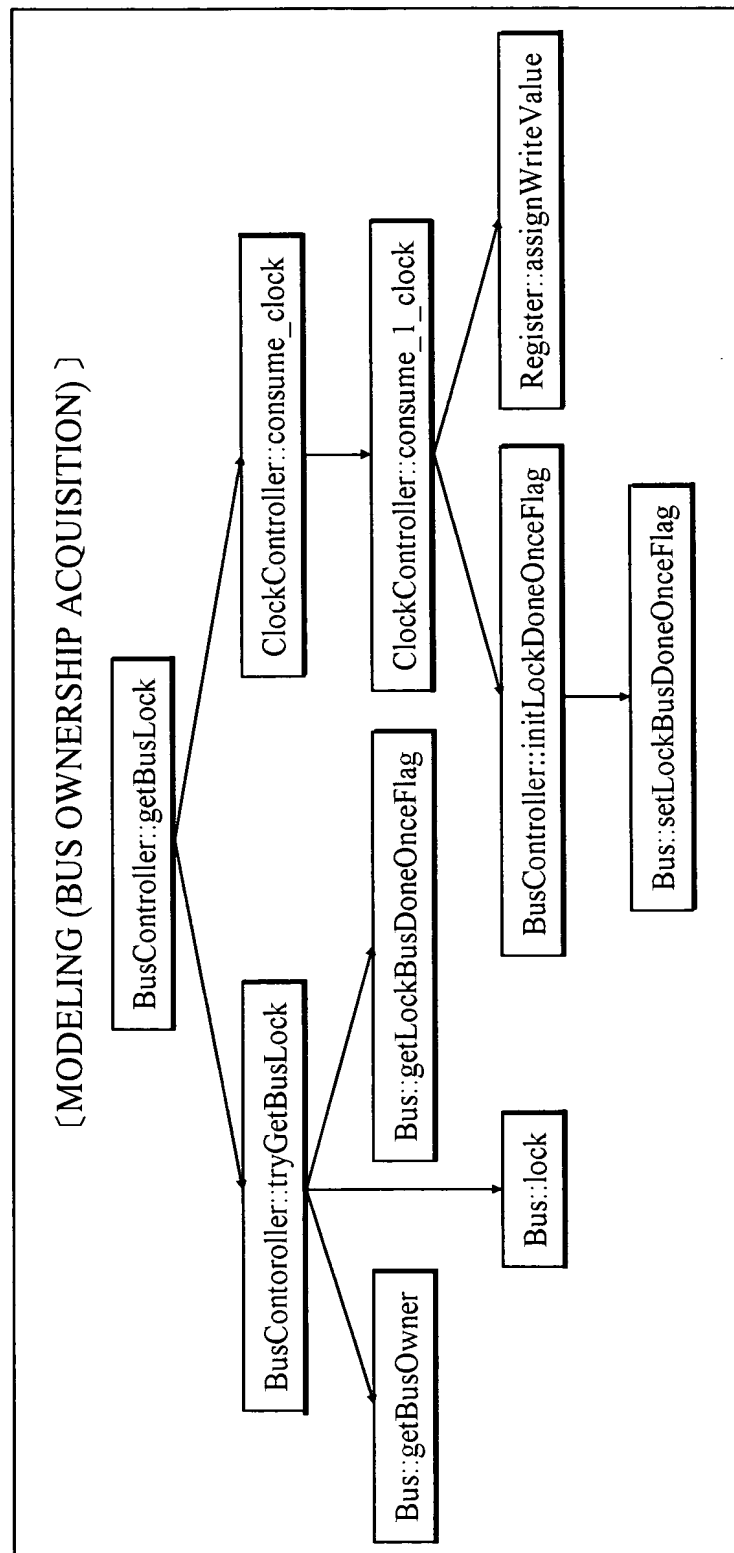


FIG. 7**[MODELING (BUS OWNERSHIP ACQUISITION)]**

```

public void getBusLock(int clock_num) {
    /* CHECK IF BUS OWNERSHIP COULD BE ACQUIRED */
    while (this.tryGetBusLock() == false) {
        /* CONSUME CLOCKS IN NUMBER OF "clock_num"
           BECAUSE BUS OWNERSHIP COULD NOT BE ACQUIRED */
        cc.consume_clock(clock_num);
    }
}

```

FIG. 8**[MODELING (BUS OWNERSHIP ACQUISITION)]**

```

private synchronized boolean tryGetBusLock() {
    /* CHECK IF BUS IS NOT CURRENTLY LOCKED, AND BESIDES,
       BUS HAS NEVER BEEN LOCKED WITHIN PERTINENT CLOCK
       (THAT IS, IF BUS ACCESS FLAG IS "false") */
    if ((bus.getBusOwner() == null) && (bus.getLockDoneOnceFlag() == false)) {
        /* LOCK BUS */
        this.bus.lock();
        /* INCREMENT NUMBER OF TIMES OF LOCKS */
        this.busycount++;
        /* RETURN AS "true", THAT BUS OWNERSHIP COULD BE
           ACQUIRED */
        return true;
    } else if (bus.getBusOwner() == Thread.currentThread()) {
        /* LOCK BUS AGAIN IN CASE WHERE LOCK REQUEST HAS BEEN
           FURTHER MADE BY THREAD WHICH IS CURRENTLY LOCKING
           BUS, AND HENCE, INCREMENT NUMBER OF TIMES OF LOCKS */
        this.busycount++;
        /* RETURN AS "true", THAT BUS OWNERSHIP COULD BE ACQUIRED */
        return true;
    } else {
        /* RETURN AS "false", THAT BUS OWNERSHIP COULD NOT BE
           ACQUIRED */
        return false;
    }
}

```

FIG. 9

[MODELING (BUS OWNERSHIP RELEASE)]	
METHOD NAME	
freeBusLock	BUS OWNERSHIP IS RELEASED BY CALLING "unlock" METHOD, "setLockBusDoneOnceFlag" IS CALLED BY SETTING ARGUMENT AT "true", AND "consume_clock" IS CALLED SO AS TO CONSUME CLOCKS IN NUMBER OF "clock_num"
unlock	IF OBJECT CURRENTLY LOCKING BUS IS "Thread" CURRENTLY EXECUTING "Bus class" OBJECT ("lock" METHOD) IS CHECKED, AND IF SO, OBJECT CURRENTLY LOCKING BUS IS SET AT "null".
consume_clock	"consume_1_clock" IS CALLED.
consume_1_clock	AS ALREADY EXPLAINED ON CLOCK SYNCHRONIZATION MECHANISM.
initLockBusDoneOnceFlag	"setLockBusDoneOnceFlag" IS CALLED BY SETTING ARGUMENT AT "false".
setLockBusDoneOnceFlag	ARGUMENT IS ASSIGNED TO BUS ACCESS FLAG.
assignWriteValue	IN CASE WHERE RIGHT ACCESS TO SHARED REGISTER IS EXISTENT, REGISTER ASSIGNMENT IS EXECUTED BEFORE PROCEEDING TO NEXT CLOCK.

FIG. 10

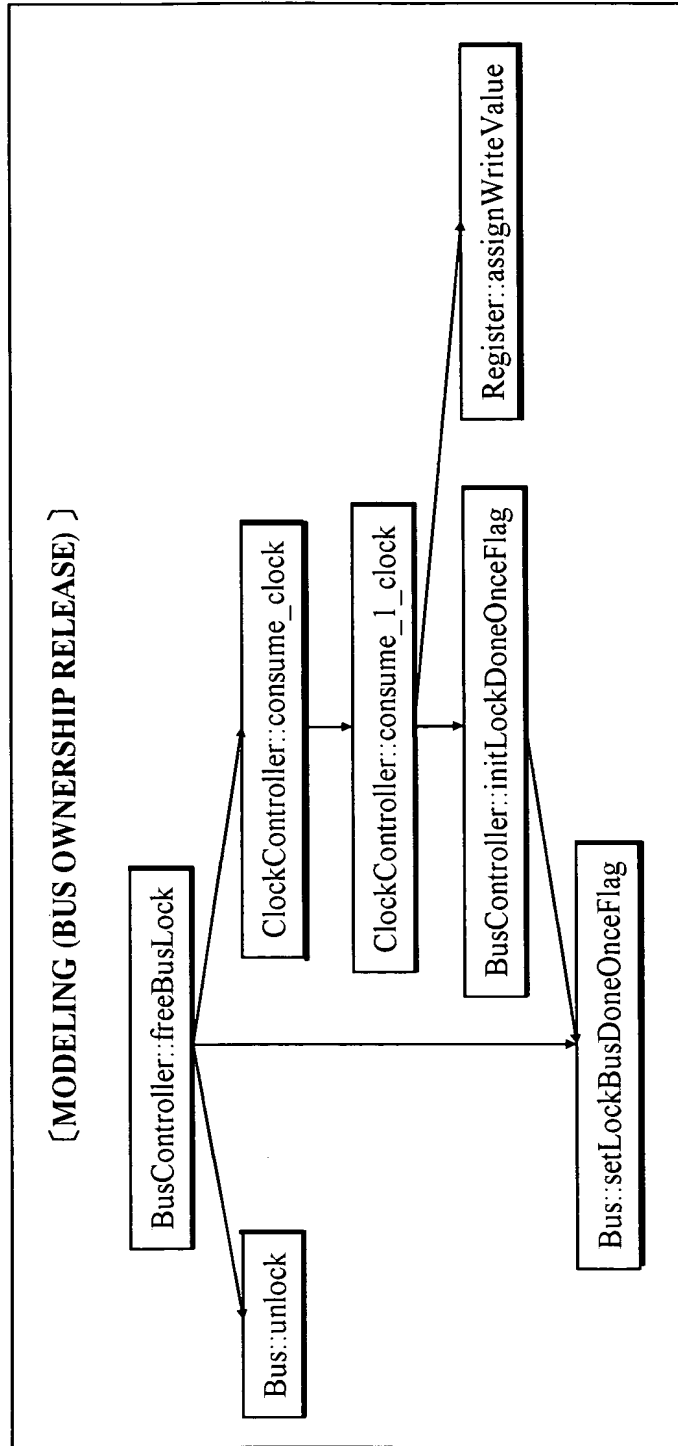


FIG. 11**[MODELING (BUS OWNERSHIP RELEASE)]**

```

public void freeBusLock(int clock_num) {
    synchronized (this) {
        /* CHECK IF CALLED THREAD IS THREAD WHICH IS LOCKING
        BUS */
        if (this.bus.getBusOwner() == Thread.currentThread()) {
            /* DECREMENT NUMBER OF TIMES OF LOCKS */
            this.busycount--;
            /* CHECK IF DECREMENTED RESULT IS "0" */
            if (this.busycount == 0) {
                /* UNLOCK BUS */
                this.bus.unlock();
            }
            /* SET BUS ACCESS FLAG AT "true" */
            this.bus.setLockDoneOnceFlag(true);
        }
    }
    /* CONSUME CLOCKS IN NUMBER OF "clock_num" */
    cc.consume_clock(clock_num);
}

```

FIG. 12**[MODELING (EXCLUSIVE SYNCHRONIZED READ)]**

```

public synchronized int sync_read(BusController bc,
                                   int index,
                                   int clock_num) {

    /* ACQUIRE BUS OWNERSHIP */
    bc.getBusLock(clock_num);
    /* READ VALUE OF DESIGNATED OBJECT
    SHARED-VARIABLE */
    int read_value = this.current_value[index];
    /* RELEASE BUS OWNERSHIP */
    bc.freeBusLock(clock_num);
    /* RETURN READ VALUE */
    return read_value;
}

```

[sync_read METHOD]

FIG. 13

[MODELING (EXCLUSIVE SYNCHRONIZED READ)]

```

public void run() {
    Register other_r0 = (Register)super.access_registers.get(0);
    int read_value;
    while (true) {
        this.do_something_w_or_wo_clock_boundary1();
        read_value = other_r0.sync_read(super.bc0, 1);
        this.do_something_w_or_wo_clock_boundary2();
    }
}

```

[DESCRIPTIVE EXAMPLE IN "run()"]

FIG. 14

[MODELING (EXCLUSIVE SYNCHRONIZED READ)]

```

public int sync_burst_read(BusController bc, int index,
                           int clock_num) {
    /* REPEAT LOCK EVERY CALL AS LONG AS LOCK
       CONDITION IS FULFILLED */
    bc.getBusLock(clock_num);
    /* READ VALUE OF DESIGNATED OBJECT SHARED-VARIABLE */
    int read_value = this.current_value[index];
    /* RETURN READ VALUE */
    return read_value;
}

```

[sync_burst_read METHOD]

FIG. 15

[MODELING (EXCLUSIVE SYNCHRONIZED READ)]

```

public void endBurstAccess(BusController bc, int clock_num) {
    bc.freeBurstBusLock(clock_num);
}

```

[endBurstAccess METHOD]

FIG. 16

[MODELING (EXCLUSIVE SYNCHRONIZED READ)]

```

public void freeBurstBusLock(int clock_num) {
    synchronized (this) {
        /* CHECK IF CALLED THREAD IS THREAD WHICH IS
           LOCKING BUS */
        if (this.bus.getBusOwner() == Thread.currentThread()) {
            /* RESET NUMBER OF TIMES OF LOCKS TO "0" */
            this.busycount = 0;
            /* UNLOCK BUS */
            this.bus.unlock();
        }
        /* SET BUS ACCESS FLAG AT "true" */
        this.bus.setLockDoneOnceFlag(true);
    }
    /* CONSUME CLOCKS IN NUMBER OF "clock_num" */
    cc.consume_clock(clock_num);
}

```

[freeBurstBusLock METHOD]

FIG. 17**[MODELING (EXCLUSIVE SYNCHRONIZED READ)]**

```

public void run() {
    Register other_r1 = (Register)super.access_registers.get(1);
    int read_value[10]
    while (true) {
        this.do_something_w_or_wo_clock_boundary1();
        /* BURST READS (10 TIMES OF SUCCESSIVE READS) */
        synchronized (this) {
            int i;
            for (i=0; i<10; i++) {
                read_value[i] = other_r1.sync_burst_read(super.bc, i, 1));
                super.cc.consume_clock(1);
                this.do_something_wo_clock_boundary1();
            }
            other_r1.sync_burst_read(super.bc, i, 1);
            other_r1.endBurstAccess(super.bc, 1);
            this.do_something_wo_clock_boundary2();
        }
        this.do_something_w_or_wo_clock_boundary2();
    }
}

```

[DESCRIPTIVE EXAMPLE IN "run()"]

FIG. 18

[MODELING (EXCLUSIVE SYNCHRONIZED WRITE)]

```

public synchronized void sync_write(BusController bc,
                                     int write_value,
                                     int index, int
clock_num) {
    /* ACQUIRE BUS OWNERSHIP */
    bc.getBusLock(clock_num);
    /* HOLD WRITE VALUE FOR DESIGNATED OBJECT
    SHARED-VARIABLE */
    this.write_value = write_value;
    /* NOTIFY USER OF ACCESSED ARRAY */
    this.update_index = index;
    /* NOTIFY USER THAT WRITING INTO SHARED
    VARIABLE HAS BEEN DONE, AND EXECUTE
    WRITING INTO SHARED VARIABLE
    IMMEDIATELY BEFORE TRANSITION TO NEXT
    CLOCK BY "consume_1_clock" */
    this.write_access = true;
    /* RELEASE BUS OWNERSHIP */
    bc.freeBusLock(clock_num);
}

```

[sync_write METHOD]

FIG. 19

[MODELING (EXCLUSIVE SYNCHRONIZED WRITE)]

```

public void run() {
    Register other_r0 = (Register)super.access_registers.get(0);
    int write_value;
    while (true) {
        this.do_something_w_or_wo_clock_boundary1();
        other_r0.sync_write(super.bc, write_value0, 1);
        this.do_something_w_or_wo_clock_boundary2();
    }
}

```

[DESCRIPTIVE EXAMPLE IN "run()"]

FIG. 20

[MODELING (EXCLUSIVE SYNCHRONIZED WRITE)]

```
public int sync_burst_read {(BusController bc, int write_value, int index, int clock_num) {  
    /* REPEAT LOCK EVERY CALL AS LONG AS LOCK CONDITION IS FULFILLED */  
    bc.getBusLock(clock_num);  
    /* HOLD WRITE VALUE FOR DESIGNATED OBJECT SHARED-VARIABLE */  
    this.write_value = write_value;  
    /* NOTIFY USER THAT WRITING INTO SHARED VARIABLE HAS BEEN DONE, AND  
    EXECUTE WRITING INTO SHARED VARIABLE IMMEDIATELY BEFORE TRANSITION  
    TO NEXT CLOCK BY "consume_1_clock" */  
    this.write_access = true;  
}
```

sync_burst_write METHOD

FIG. 21**[MODELING (EXCLUSIVE SYNCHRONIZED WRITE)]**

```

public void run() {
    Register other_r0 = (Register)super.access_registers.get(0);
    int write_value[10]
    while (true) {
        this.do_something_w_or_wo_clock_boundary1();
        /* BURST WRITES (10 TIMES OF SUCCESSIVE WRITES) */
        synchronized (this) {
            int i;
            for (i=0; i<10; i++) {
                other_r0.sync_burst_write(super.bc, write_value[i], 1));
                super.cc.consume_clock(1);
                this.do_something_wo_clock_boundary1();
            }
            other_r1.sync_burst_write(super.bc, write_value[i], 1);
            other_r1.endBurstAccess(super.bc, 1);
            this.do_something_wo_clock_boundary2();
        }
        this.do_something_w_or_wo_clock_boundary2();
    }
}

```

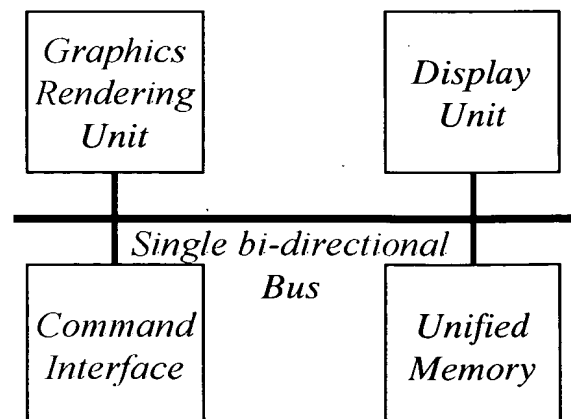
DESCRIPTIVE EXAMPLE IN "run()"**FIG. 22**

FIG. 23

[INSTALLATION EXAMPLE OF "run()" METHOD (COMMAND INTERFACE)]

```

public void run() {
    Register mem_con_reg = (Register)super.access_registers.get(2);
    int[] drawing_commands = {100, 101, 102, 103, 104};
    int dcom_index = 0;
    int com_flag_0 = 0;
    int com_flag_1 = 0;
    while (true) {
        /* EXECUTE COMMAND ACCEPTANCE IN CASE WHERE EXTERNAL
        INPUT "write_req" SIGNAL EXISTS AND WHERE "wait_flag" SIGNAL
        IS "false", AND MERELY CONSUME CLOCK OTHERWISE */
        if (this.getWriteReqSignal() && (this.wait_flag == false)) {
            /* EXECUTE ACCEPTANCE OF COMMANDS */
            this.input_command = drawing_commands[dcom_index++];
            /* CONSUME PREDETERMINED NUMBER OF CLOCKS BEING
            3 CLOCKS HERE */
            super.cc.consume_clock(3);
            /* SET "wait_flag" VARIABLE AT "true" */
            this.wait_flag = true;
            /* SET "wait" OUTPUT SIGNAL AT "true", AND CONSUME ONE CLOCK*/
            this.wait_output_signal = true;
            super.cc.consume_clock( 1);
        }
    }
}

```

DESCRIPTIONS INSERTED
FOR SIMULATION

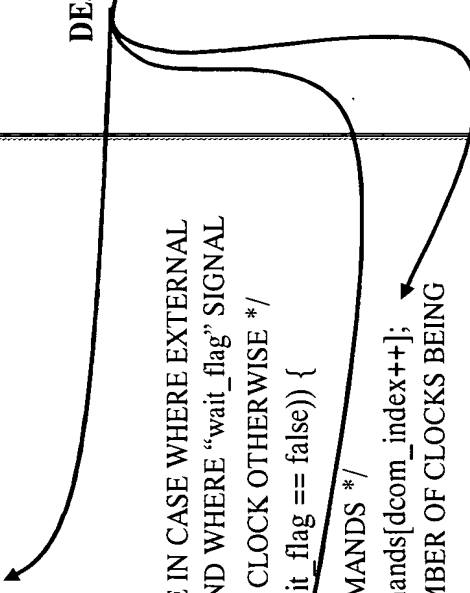


FIG. 24**[INSTALLATION EXAMPLE OF "run0" METHOD (COMMAND INTERFACE)]**

```

/* UPDATE VALUE OF CORRESPONDING RENDERING COMMAND IN CASE WHERE VALUE OF COMMAND
FLAG IS "0"; GIVE PREFERENCE TO RENDERING COMMAND CORRESPONDING TO "0" IN CASE WHERE
VALUES OF BOTH COMMAND SIGNALS 0 AND 1 ARE "0"; WAIT FOR PREDETERMINED NUMBER OF CLOCKS
AND EXECUTE RENDERING COMMAND AGAIN IN CASE WHERE VALUES OF BOTH COMMAND SIGNALS 0
AND 1 ARE "1". */
while (true) {
    synchronized (this) {
        /* BURST-READ COMMAND FLAGS 0 AND 1 */
        com_flag_0 = mem_con_reg.sync_burst_read(super.bc, 0, 1);
        super.cc.consume_clock(1); //CONSUME CLOCK
        com_flag_1 = mem_con_reg.sync_burst_read(super.bc, 2, 1);
        super.cc.consume_clock(1); // CONSUME CLOCK
        if (com_flag_0 == 0) {
            /* IN CASE WHERE VALUES OF COMMAND FLAG 0 IS "0", OR WHERE BOTH VALUES OF
            COMMAND FLAGS 0 AND 1 ARE "0" */
            /* WRITE COMMAND INPUT TO RENDERING COMMAND 0 */
            mem_con_reg.sync_burst_write(super.bc, input_command, 1, 1);
            super.cc.consume_clock(1);
            /* SET VALUE OF COMMAND FLAG 0 AT "1" */
            mem_con_reg.sync_burst_write(super.bc, 1, 0, 1);
            /* END BURST MODE (INCLUDING CLOCK CONSUMPTION) */
            mem_con_reg.endBurstAccess(super.bc, 1);
            break;

```

FIG. 25

[INSTALLATION EXAMPLE OF "run()" METHOD (COMMAND INTERFACE)]

```

    } else if (com_flag_1 == 0) {
        /* IN CASE WHERE VALUE OF COMMAND FLAG 1 IS "0" */
        /* WRITE COMMAND INPUT TO RENDERING COMMAND 1 */
        mem_con_reg.sync_burst_write(super.bc, input_command, 3, 1);
        super.cc.consume_clock(1);
        /* SET VALUE OF COMMAND FLAG 1 AT "1" */
        mem_con_reg.sync_burst_write(super.bc, 1, 2, 1);
        /* END BURST MODE (INCLUDING CLOCK CONSUMPTION) */
        mem_con_reg.endBurstAccess(super.bc, 1);
        break;
    } else {
        /* IN CASE WHERE VALUES OF BOTH COMMAND FLAGS
           0 AND 1 ARE "1"*/
        /* END BURST MODE (INCLUDING CLOCK CONSUMPTION) */
        mem_con_reg.endBurstAccess(super.bc, 1);
        /* WAIT FOR PREDETERMINED NUMBER OF CLOCKS BEING
           3 CLOCKS HERE*/
        super.cc.consume_clock(3);
    }
} // end of synchronized
} // end of nested while-loop

```

FIG. 27

[INSTALLATION EXAMPLE OF "run()" METHOD (UNIFIED MEMORY)]

```

public void run() {
    /* INSTANCE REGISTERS WITHIN DEVICES ON BUS AS WRITE DATA
       INTO UNIFIED MEMORY */
    // GROUP OF REGISTERS WITHIN GRAPHICS RENDERING UNIT
    Register renderer_reg = (Register)super.access_registers.get(0);
    // GROUP OF REGISTERS WITHIN DISPLAY UNIT
    Register display_reg = (Register)super.access_registers.get(1);
    // GROUP OF REGISTERS WITHIN COMMAND INTERFACE
    Register com_fetch_reg = (Register)super.access_registers.get(2);
    /* ACTUALLY EXECUTE NOTHING IN ORDER TO SIMPLIFY MODEL */
    while (true) {
        /* CONSUME ONE CLOCK */
        super.cc.consume_clock(1);
    }
}

```

FIG. 26

[INSTALLATION EXAMPLE OF "run()" METHOD (COMMAND INTERFACE)]

```

/* SET "wait_flag" VARIABLE AT "false" */
this.wait_flag = false;
/* SINCE "wait_flag" VARIABLE HAS BECOME "false", RESET "wait" OUTPUT SIGNAL
   TO "false" AND CONSUME ONE CLOCK */
this.wait_output_signal = false;
super.cc.consume_clock(1);
/* IN CASE WHERE RENDERING COMMAND ARRAYS HAVE BEEN USED TO LAST,
   RESET INDEX TO FIRST */
if (dcom_index == drawing_commands.length) {
    dcom_index = 0;
}
} else {
    super.cc.consume_clock(1);
}
} // end of while-loop
}

```

DESCRIPTORS
INSERTED
FOR SIMULATION

FIG. 28

[INSTALLATION EXAMPLE OF "run()" METHOD
(GRAPHICS RENDERING UNIT)]

```
public void run() {
    Register mem_con_reg = (Register)super.access_registers.get(2);
    int[] rendering_result = new int[6];
    int current_command = 0;
    int read_data = 0;
    int com_flag_0 = 0;
    int com_flag_1 = 0;
    while (true) {
        /* WAIT STATUS */
        super.cc.consume_clock(3);
        while (true) {
            synchronized (this) {
                /* BURST-READ COMMAND FLAGS 0 AND 1 */
                com_flag_0 = mem_con_reg.sync_burst_read(super.bc, 0, 1);
                super.cc.consume_clock(1); //CONSUME CLOCK
                com_flag_1 = mem_con_reg.sync_burst_read(super.bc, 2, 1);
                /* END BURST MODE (INCLUDING CLOCK CONSUMPTION) */
                mem_con_reg.endBurstAccess(super.bc, 1);
            } // end of synchronized
        }
    }
}
```

FIG. 29

[INSTALLATION EXAMPLE OF "run()" METHOD (GRAPHICS RENDERING UNIT)]

```

if (com_flag_0 == 1) {
    /* IN CASE WHERE VALUE OF COMMAND FLAG 0 IS "0", OR IN CASE WHERE
    VALUES OF BOTH COMMAND FLAGS 0 AND 1 ARE "0" */
    synchronized(this) {
        /* READ VALUE OF RENDERING COMMAND 0 */
        current_command = mem_con_reg.sync_burst_read(super.bc, 1, 1);
        /* SET "render_start" AT "true" */
        this.render_start = true;
        super.cc.consume_clock(1);
        for (int i=0; i<3; i++) {
            /* READ DATA */
            read_data = mem_con_reg.sync_burst_read(super.bc, i+4, 1);
            /* CONSUME CLOCK */
            super.cc.consume_clock(1);
            /* RENDERING */
            rendering_result[i] = this.rendering(read_data, current_command);
        } // end of for-loop
        /* SET VALUE OF COMMAND FLAG 0 AT "0" */
        mem_con_reg.sync_burst_write(super.bc, 0, 0, 1);
        /* END BURST MODE (INCLUDING CLOCK CONSUMPTION) */
        mem_con_reg.endBurstAccess(super.bc, 1);
    } // end of synchronized
}

```

FIG. 30

[INSTALLATION EXAMPLE OF "run()" METHOD
(GRAPHICS RENDERING UNIT)]

```

for (int i=3; i<6; i++) {
    /* RENDERING */
    rendering_result[i] = this.rendering(read_data, current_command);
    /* CONSUME CLOCK */
    super.cc.consume_clock(1);
} // end of for-loop
break;
} else if (com_flag_1 == 1) {
    /* IN CASE WHERE VALUE OF COMMAND FLAG 1 IS "0" */
    synchronized(this) {
        /* READ VALUE OF RENDERING COMMAND 1 */
        current_command = mem_con_reg.sync_burst_read(super.bc, 3, 1);
        /* SET "render_start" AT "true" */
        this.render_start = true;
        super.cc.consume_clock(1);
        for (int i=0; i<3; i++) {
            /* READ OUT DATA */
            read_data = mem_con_reg.sync_burst_read(super.bc, i+4, 1);
            /* CONSUME CLOCK */
            super.cc.consume_clock(1);
            /* RENDERING */
            rendering_result[i] = this.rendering(read_data, current_command);
        } // end of for-loop
    }
}

```

FIG. 31

[INSTALLATION EXAMPLE OF "run()" METHOD
(GRAPHICS RENDERING UNIT)]

```

/* SET VALUE OF COMMAND FLAG 1 AT "0" */
mem_con_reg.sync_burst_write(super.bc, 0, 2, 1);
/* END BURST MODE (INCLUDING CLOCK CONSUMPTION) */
mem_con_reg.endBurstAccess(super.bc, 1);
} // end of synchronized
for (int i=3; i<6; i++) {
/* RENDERING */
rendering_result[i] = this.rendering(read_data, current_command);
/* CONSUME CLOCK */
super.cc.consume_clock(1);
} // end of for-loop
break;
} else {
/* IN CASE WHERE VALUES OF BOTH COMMAND FLAGS
0 AND 1 ARE "1" */
/* WAIT FOR PREDETERMINED NUMBER OF CLOCKS BEING 3
CLOCKS HERE*/
super.cc.consume_clock(3);
}
} // end of nested while-loop

```

FIG. 32

[INSTALLATION EXAMPLE OF “run()” METHOD
(GRAPHICS RENDERING UNIT)]

```
/* WRITE DATA SUBJECTED TO RENDERING, INTO MEMORY */
synchronized (this) {
    for (int i=0; i<6; i++) {
        if (i == 5) {
            /* WRITE DATA */
            mem_con_reg.sync_burst_write(super.bc, rendering_result[i], i+4, 1);
            /* END BURST MODE (INCLUDING CLOCK CONSUMPTION) */
            mem_con_reg.endBurstAccess(super.bc, 1);
        } else {
            /* WRITE DATA */
            mem_con_reg.sync_burst_write(super.bc, rendering_result[i], i+4, 1);
            /* CONSUME CLOCK */
            super.cc.consume_clock(1);
        }
    } // end of for-loop
} // end of synchronized
/* END RENDERING*/
this.render_start = false;
} // end of while-loop
}
```


FIG. 33

[INSTALLATION EXAMPLE OF "run()" METHOD (DISPLAY UNIT)]

```

public void run() {
    Register mem_con_reg = (Register)super.access_registers.get(2);
    int read_data = 0;
    while (true) {
        synchronized (this) {
            for (int i=0; i<6; i++) {
                if (i == 5 ) {
                    /* READ OUT DATA */
                    read_data = mem_con_reg.sync_burst_read(super.bc, i+4, 1);
                    /* END BURST MODE (INCLUDING CLOCK CONSUMPTION) */
                    mem_con_reg.endBurstAccess(super.bc, 1);
                } else {
                    /* READ OUT DATA*/
                    read_data = mem_con_reg.sync_burst_read(super.bc, i+4, 1);
                    /* START DATA LOAD FOR DISPLAY */
                    this.display_start = true;
                    /* CONSUME CLOCK */
                    super.cc.consume_clock(1);
                }
            } // end of for-loop
        } // end of synchronized
    }
}

```

FIG. 34

[INSTALLATION EXAMPLE OF "run()" METHOD (DISPLAY UNIT)]

```

/* END DATA LOAD FOR DISPLAY */
this.display_start = false;
/* DISPLAY */
for (int i=0; i<6; i++) {
    this.display(read_data);
}
/* WAIT */
super.cc.consume_clock(3);
}
}
}

```

FIG. 35

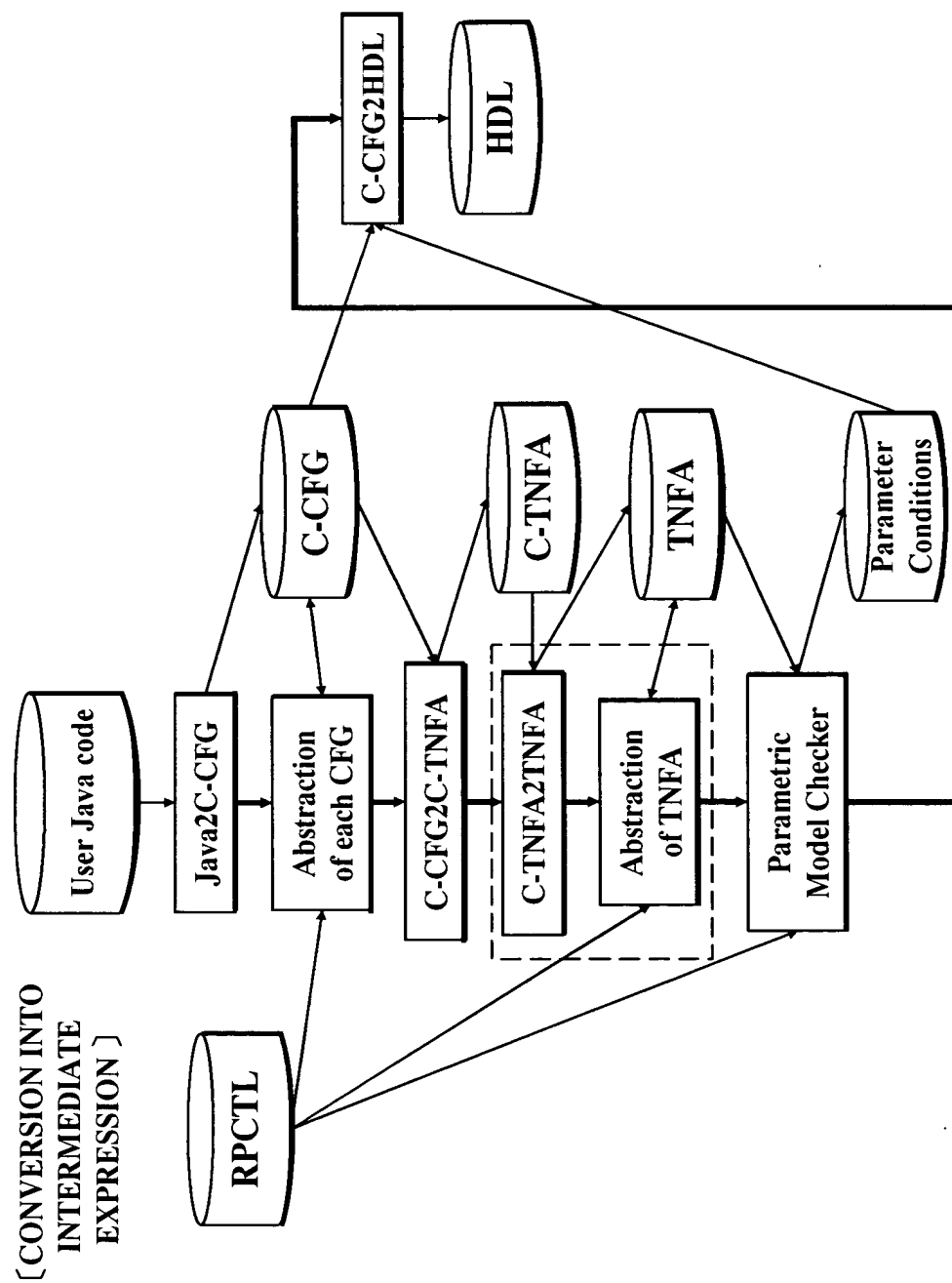


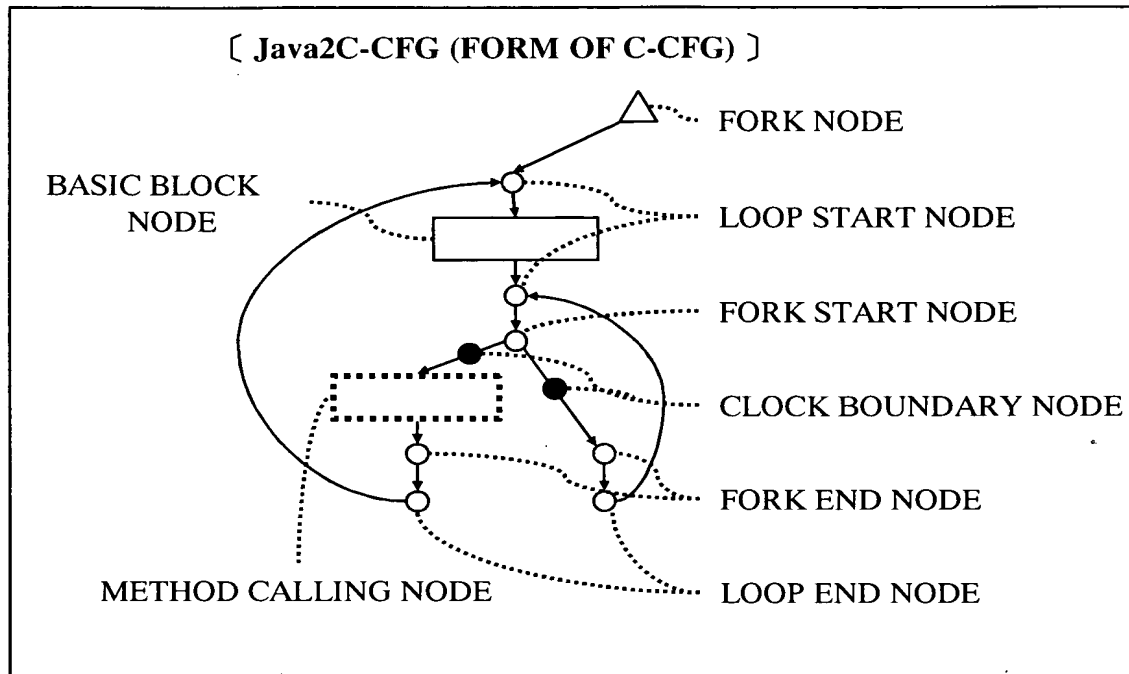
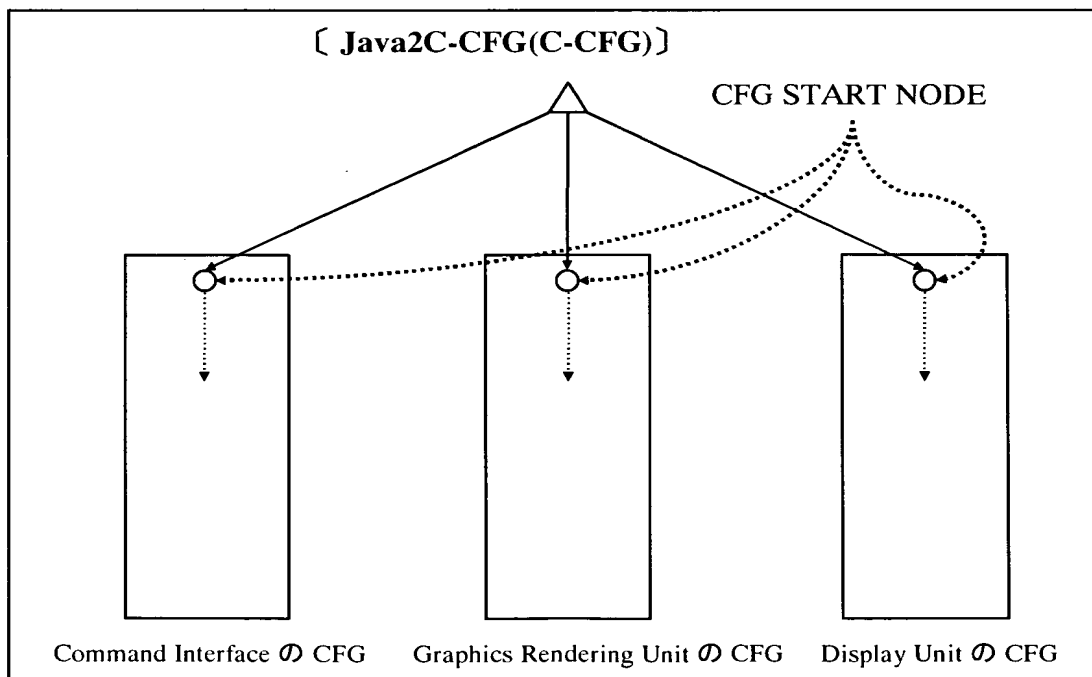
FIG. 36**FIG. 48**

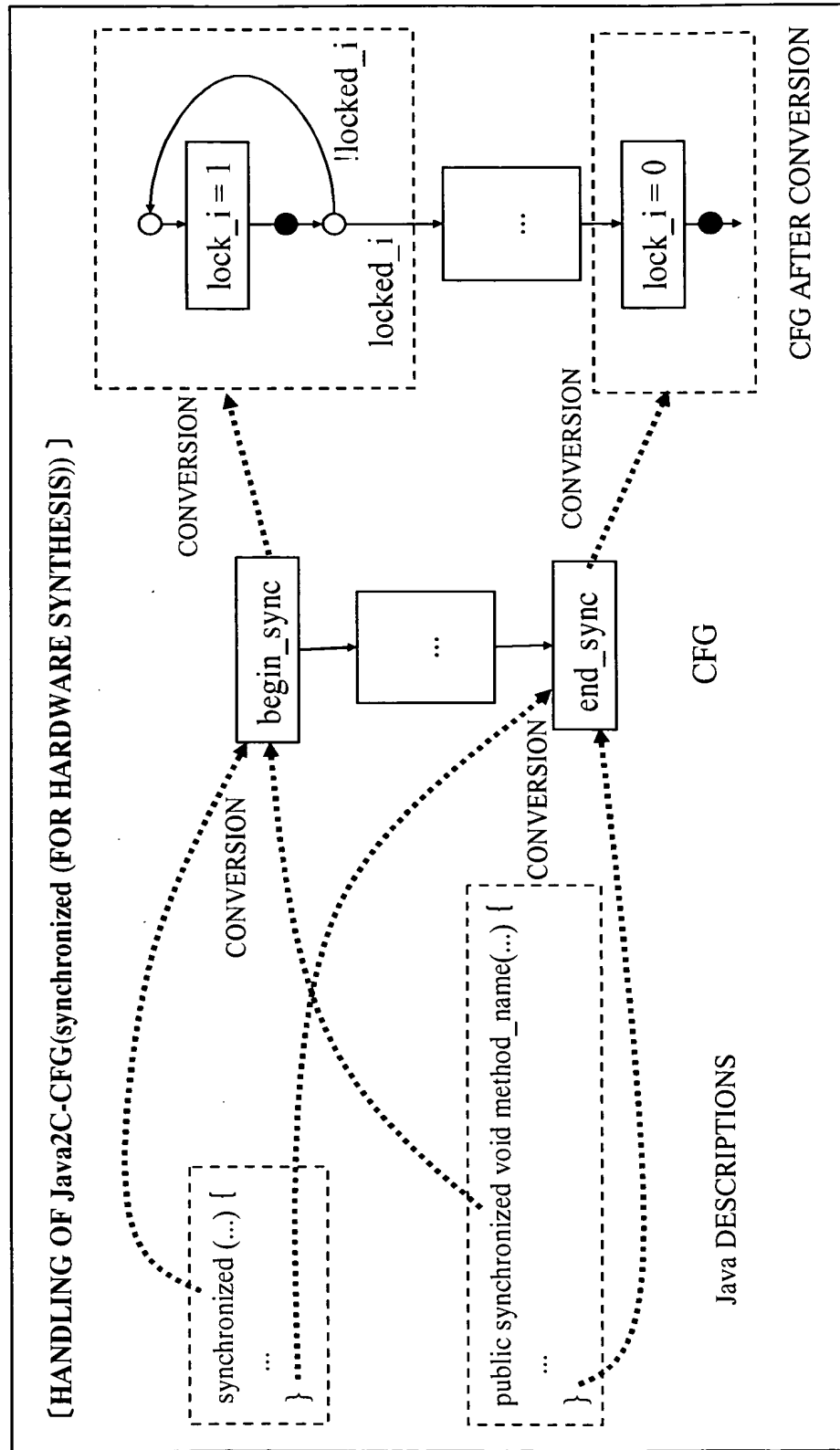
FIG. 37

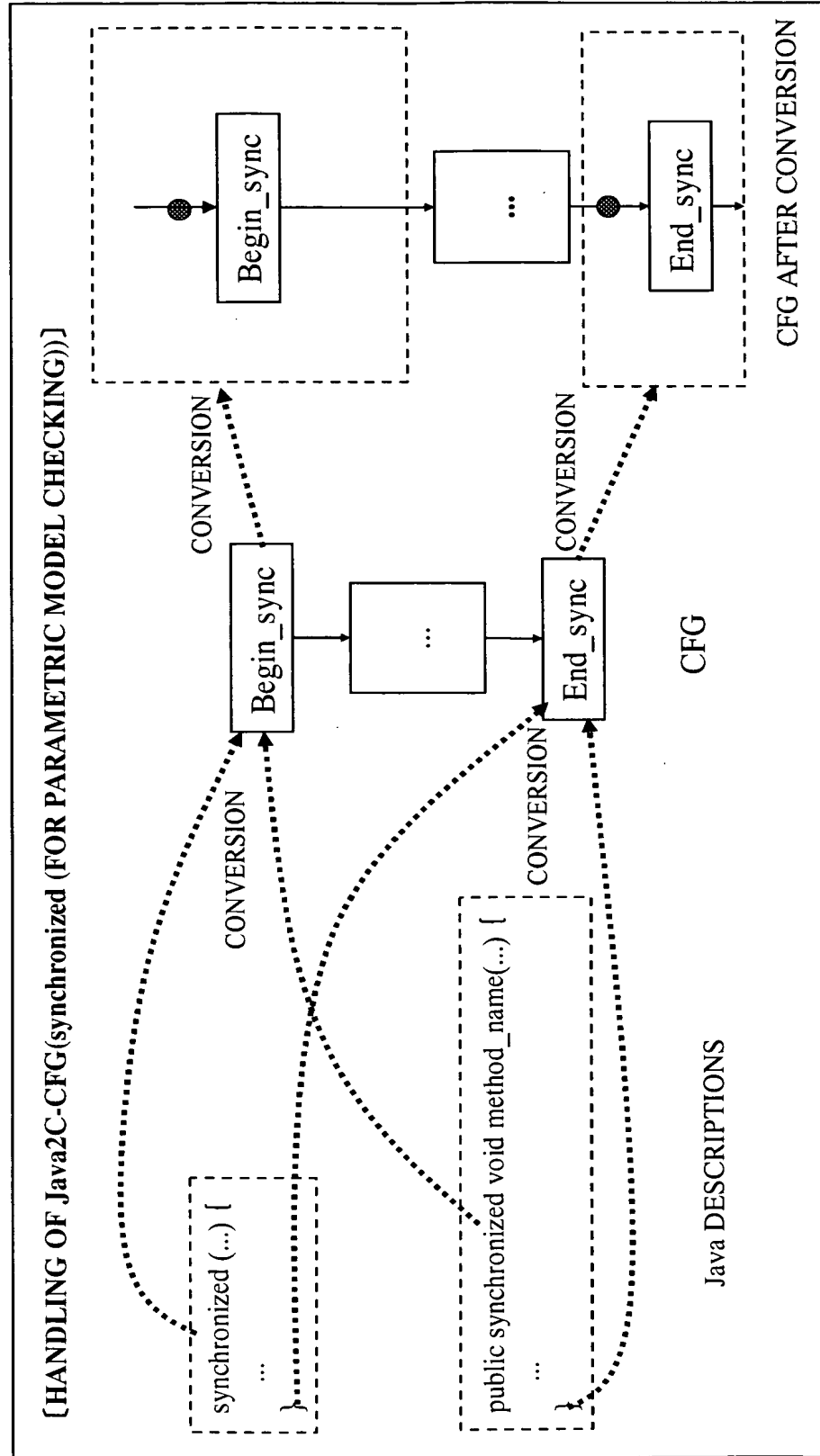
FIG. 38

FIG. 40

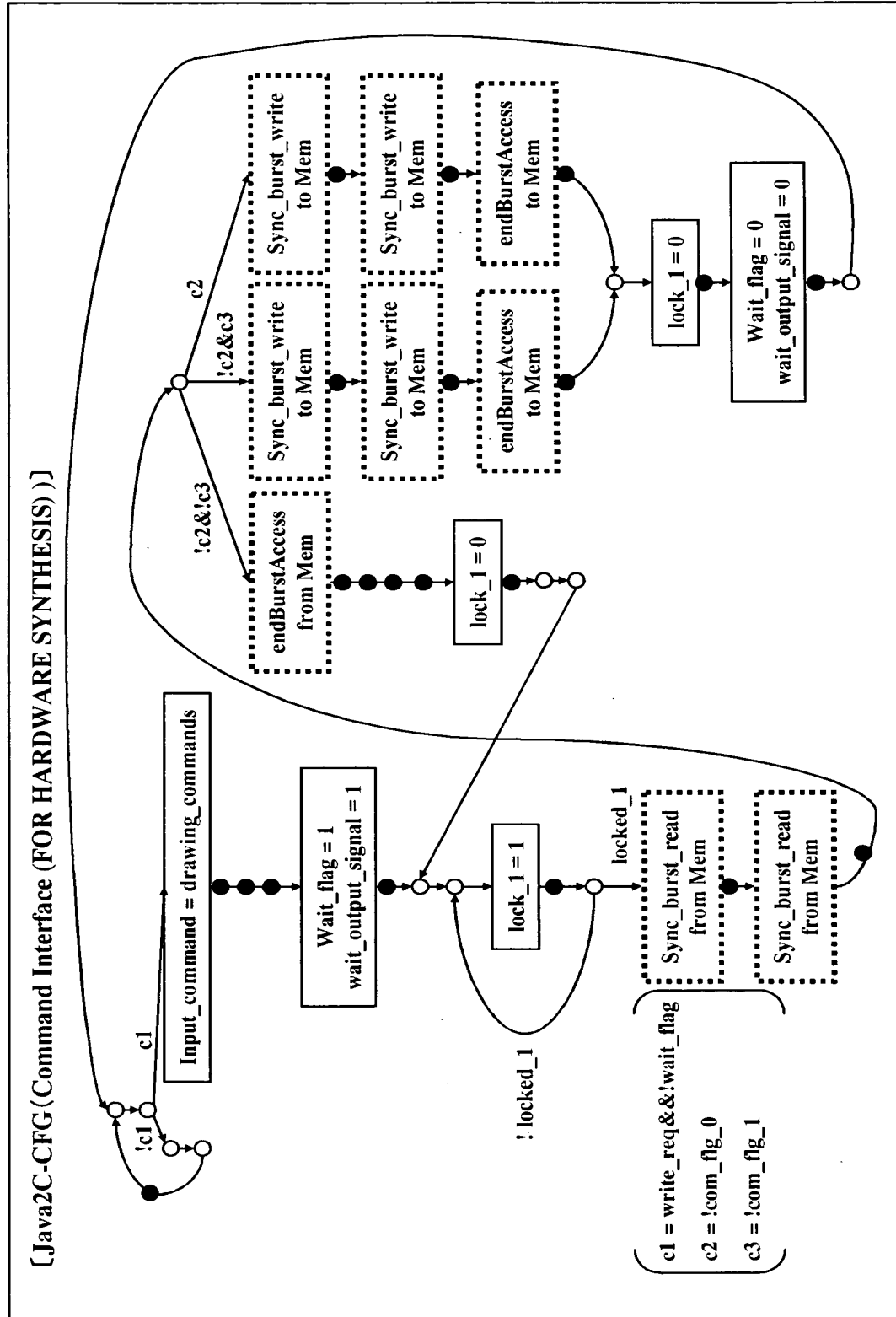


FIG. 42

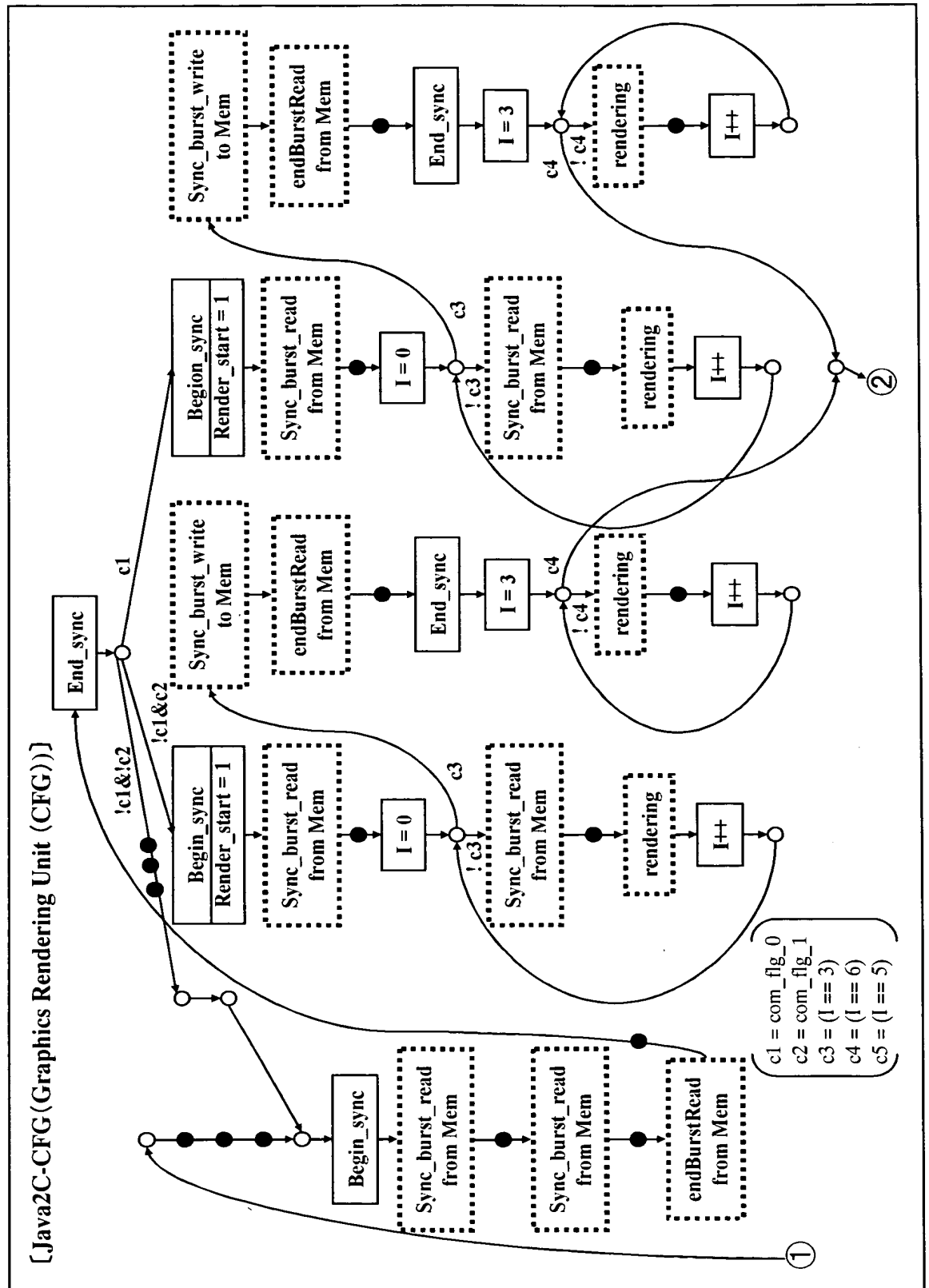
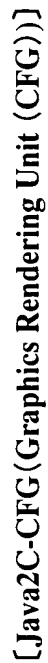


FIG. 43

[Java2C-CFG(Graphics Rendering Unit (CONTINUATION TO FIG. 42))]

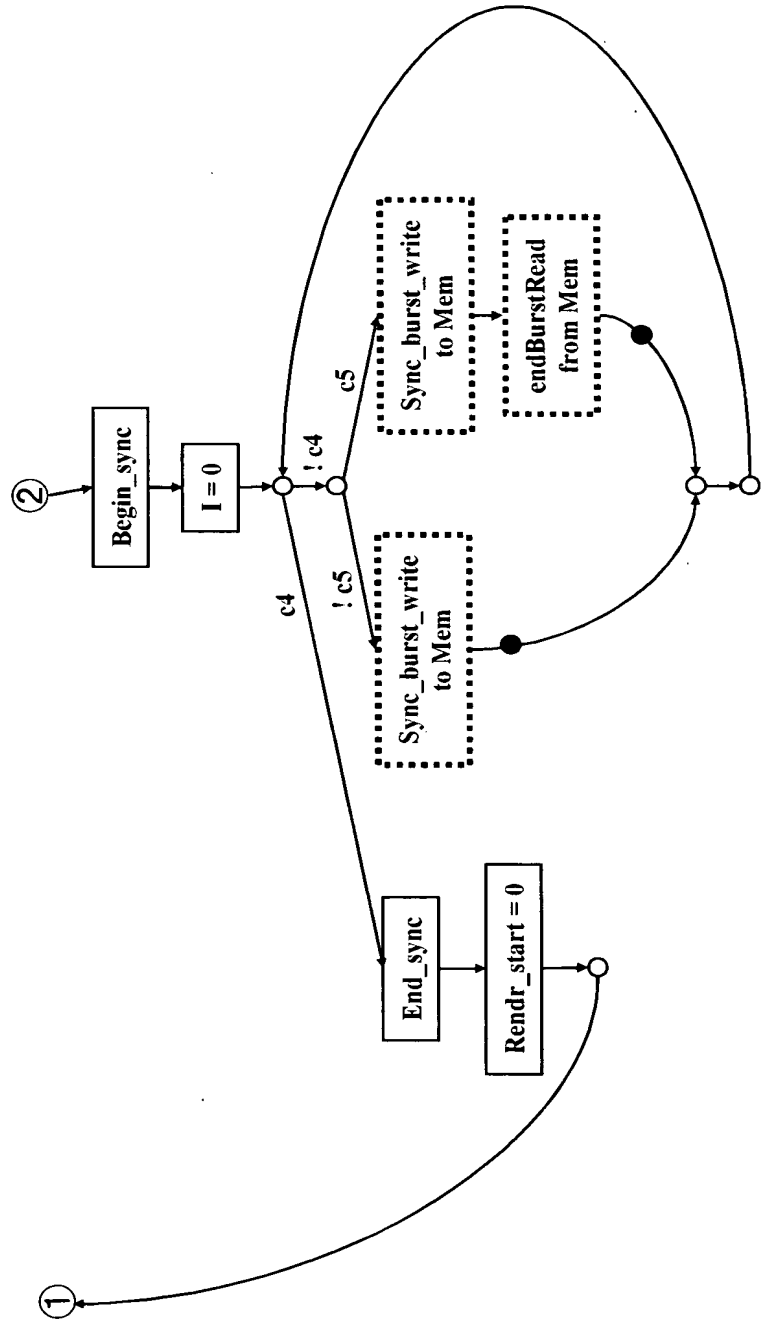


FIG. 44

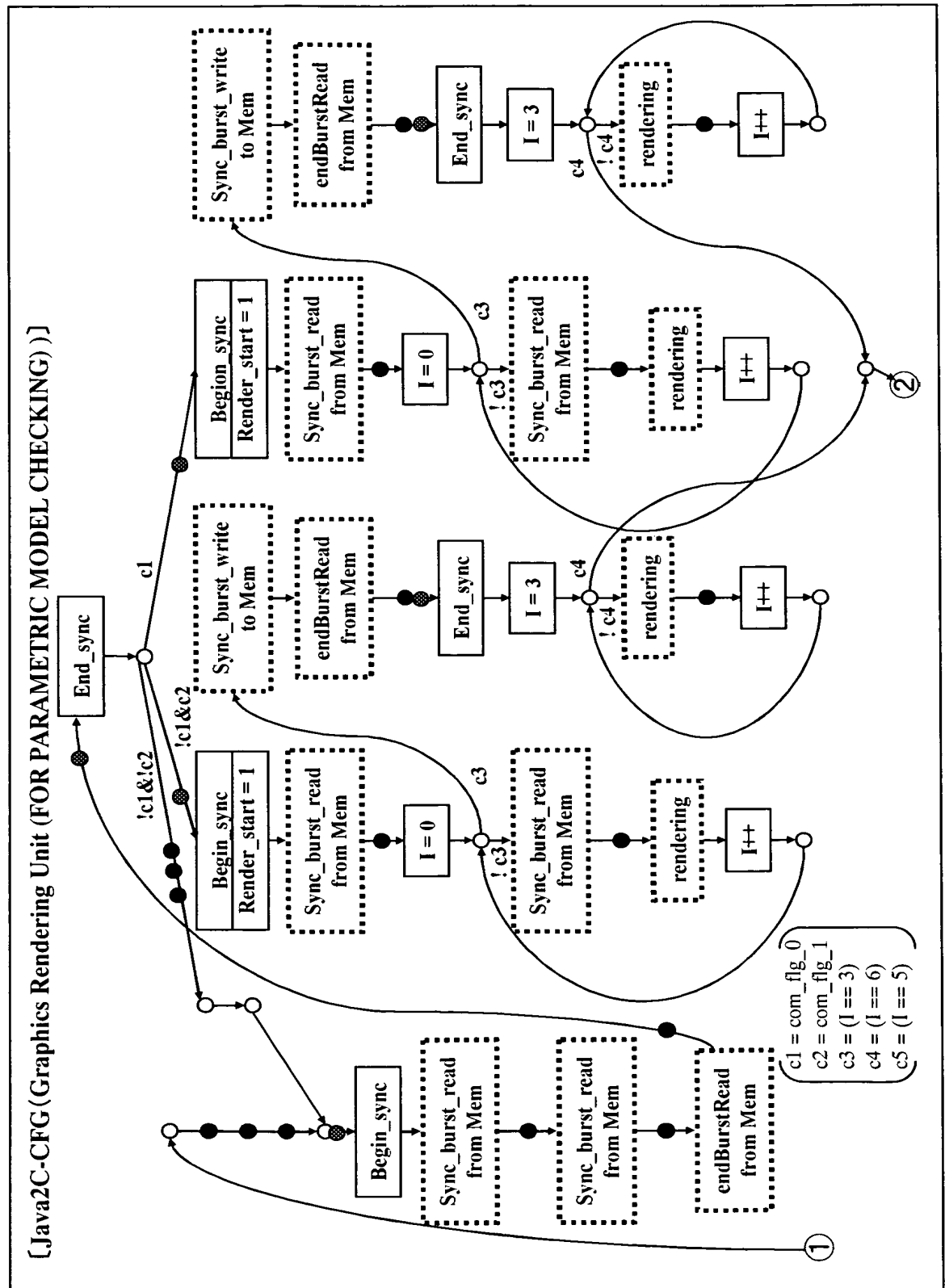


FIG. 45

[Java2C-CFG (Graphics Rendering Unit (CONTINUATION TO FIG. 44))]

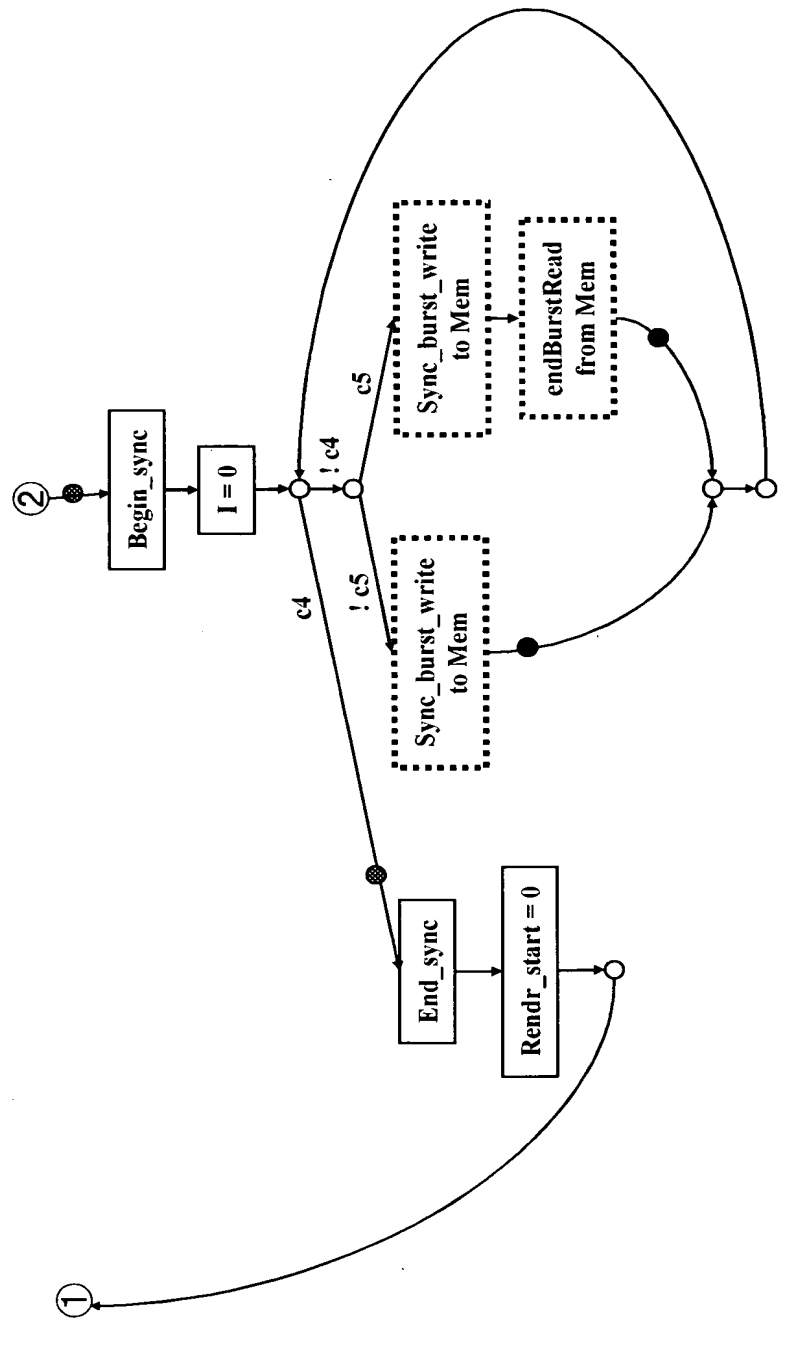


FIG. 46

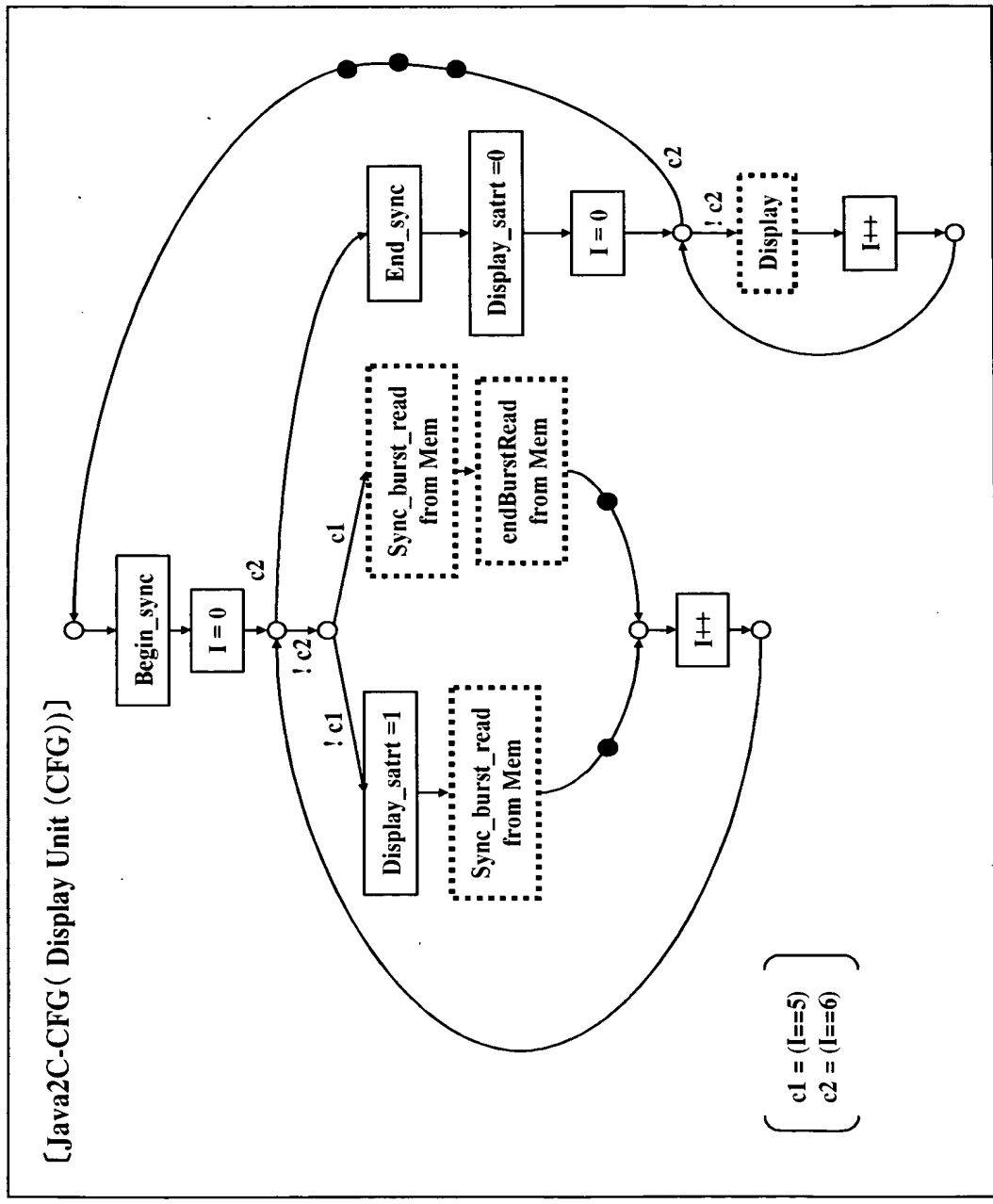


FIG. 47

[Java2C-CFG (Display Unit (FOR PARAMETRIC MODEL CHECKING))]

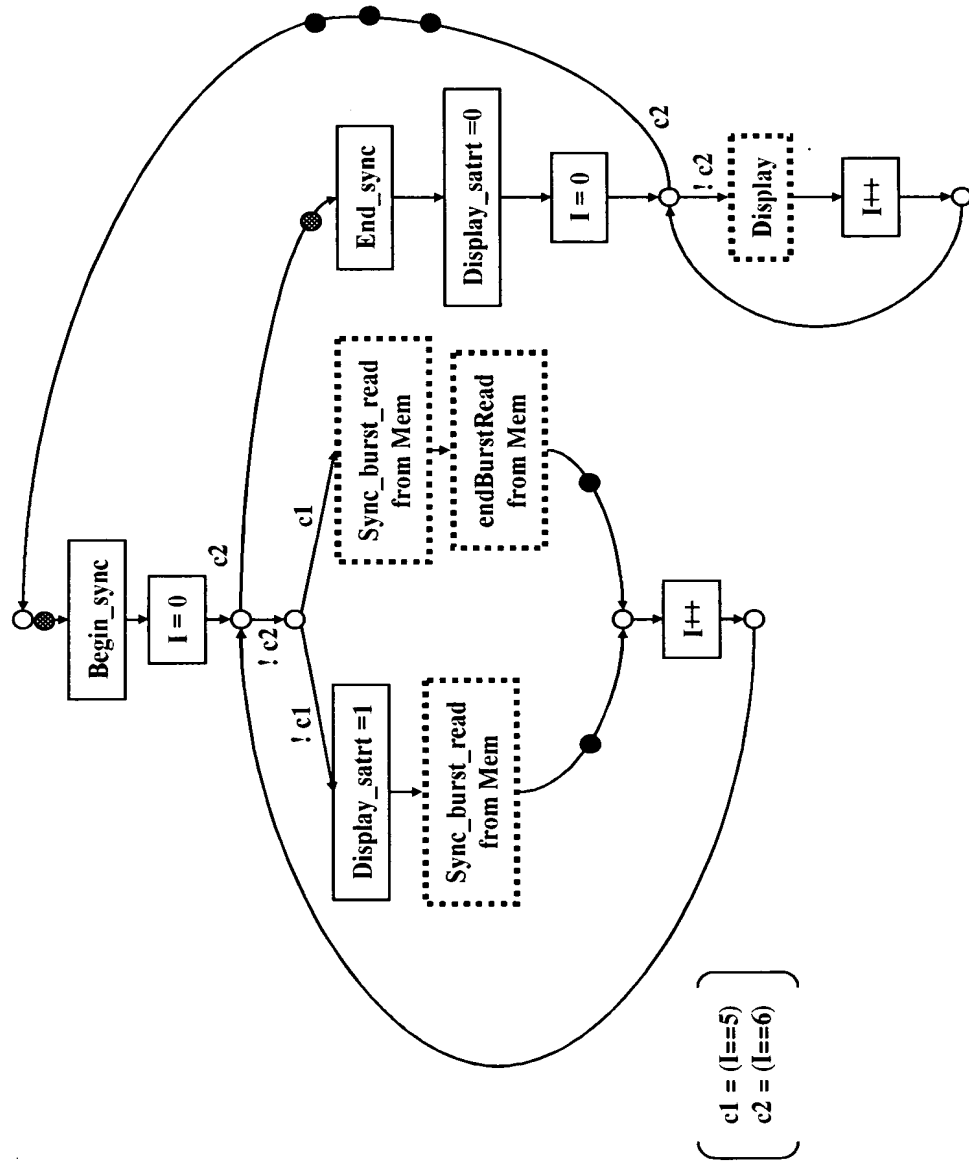


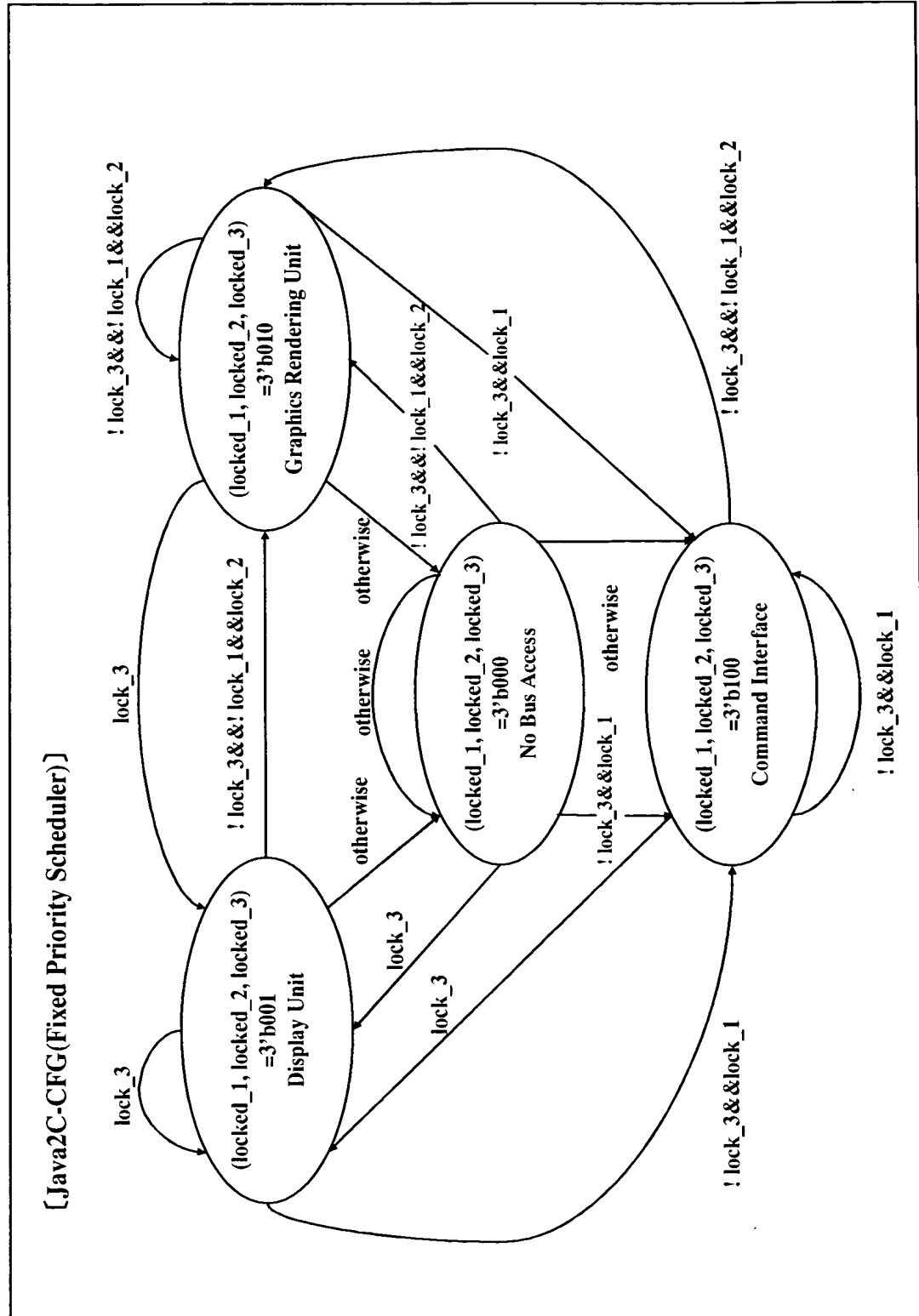
FIG. 49

FIG. 50

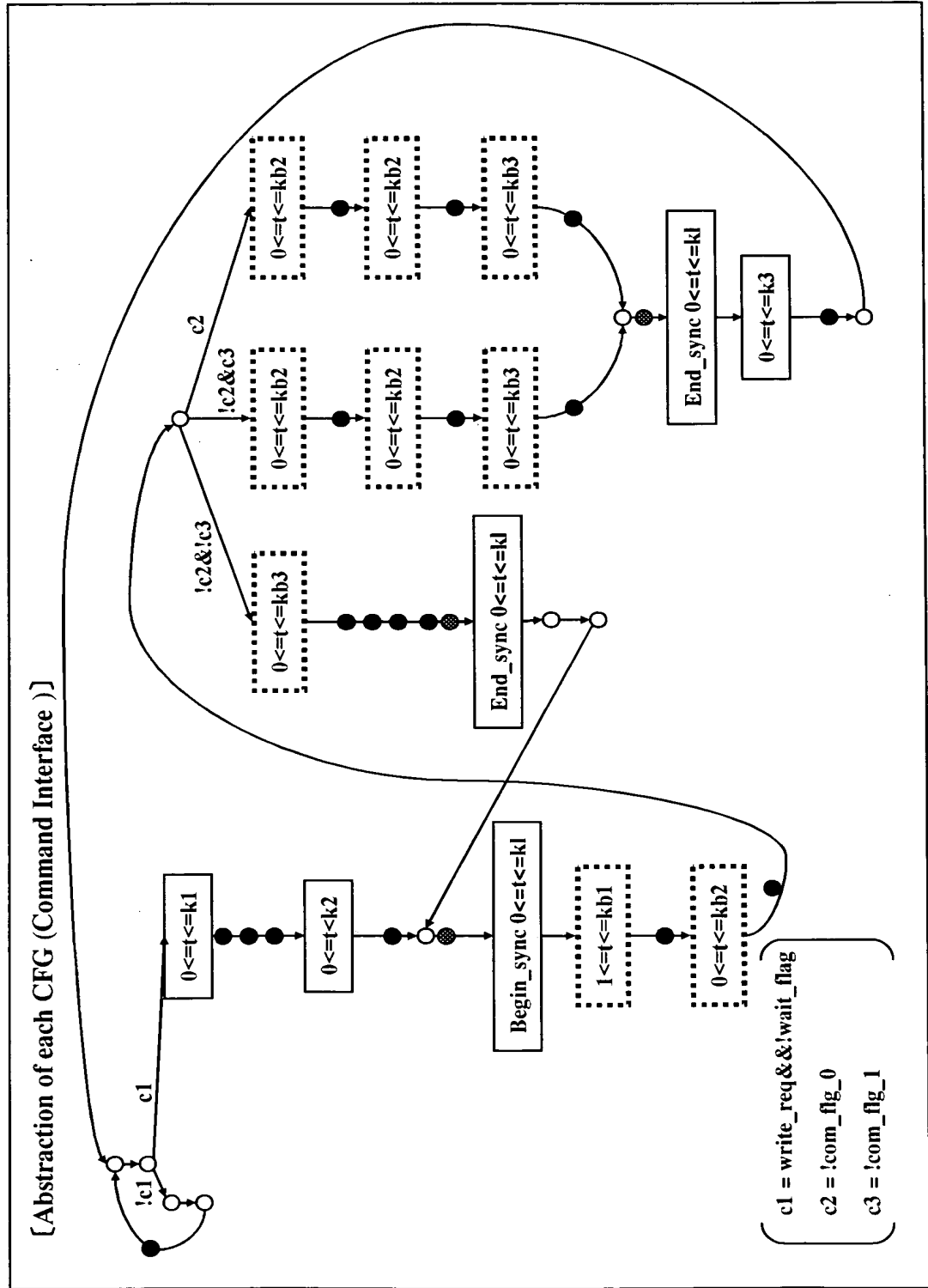


FIG. 51

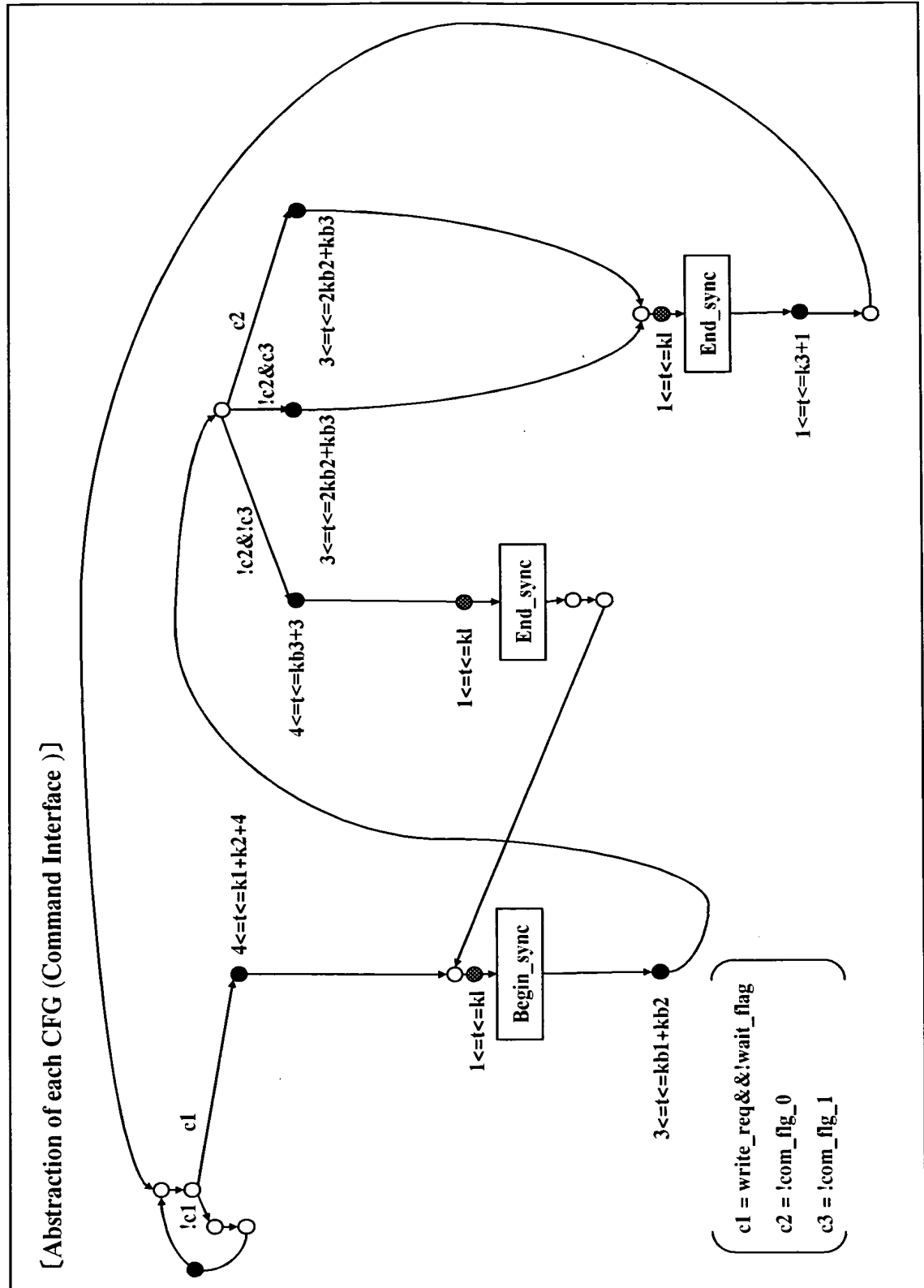


FIG. 53

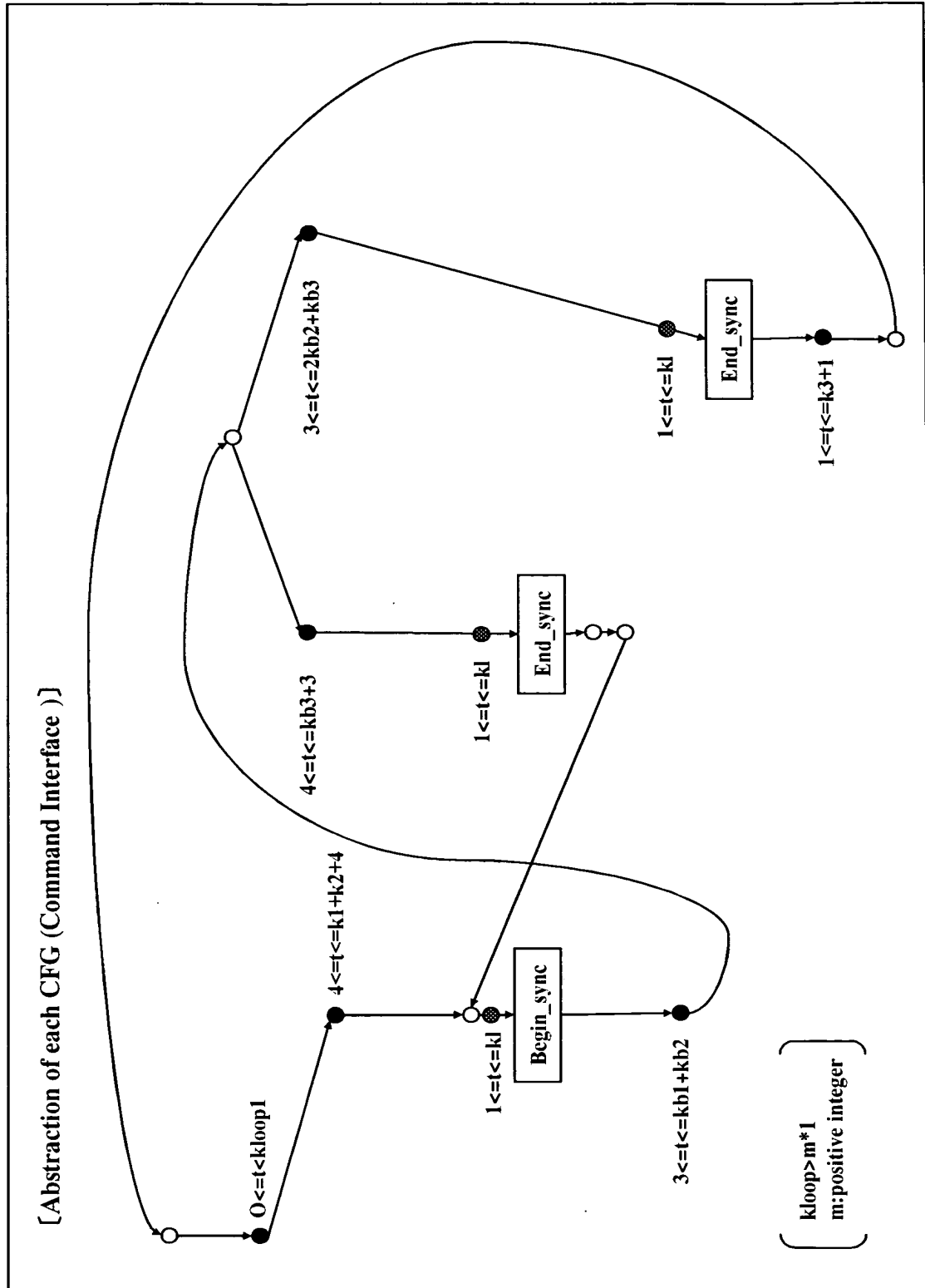


FIG. 54

[Abstraction of each CFG (Command Interface)]

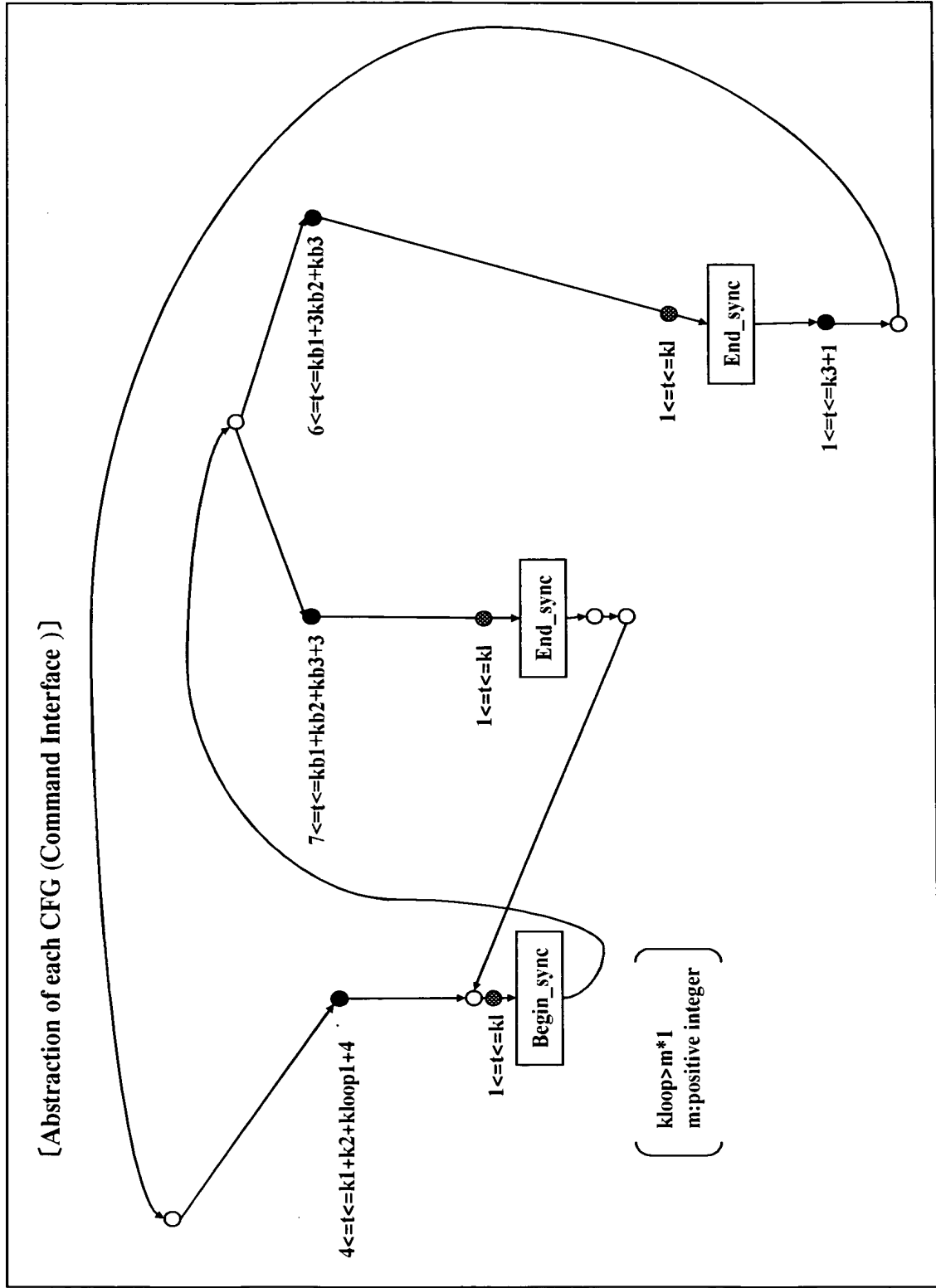


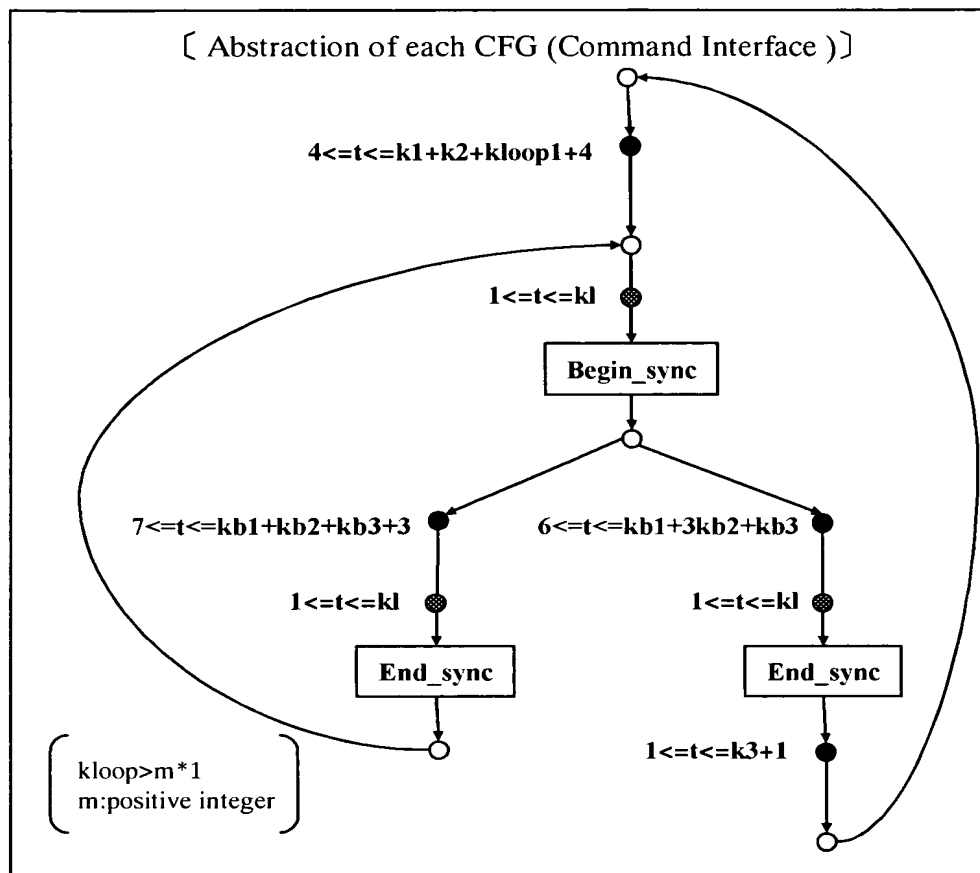
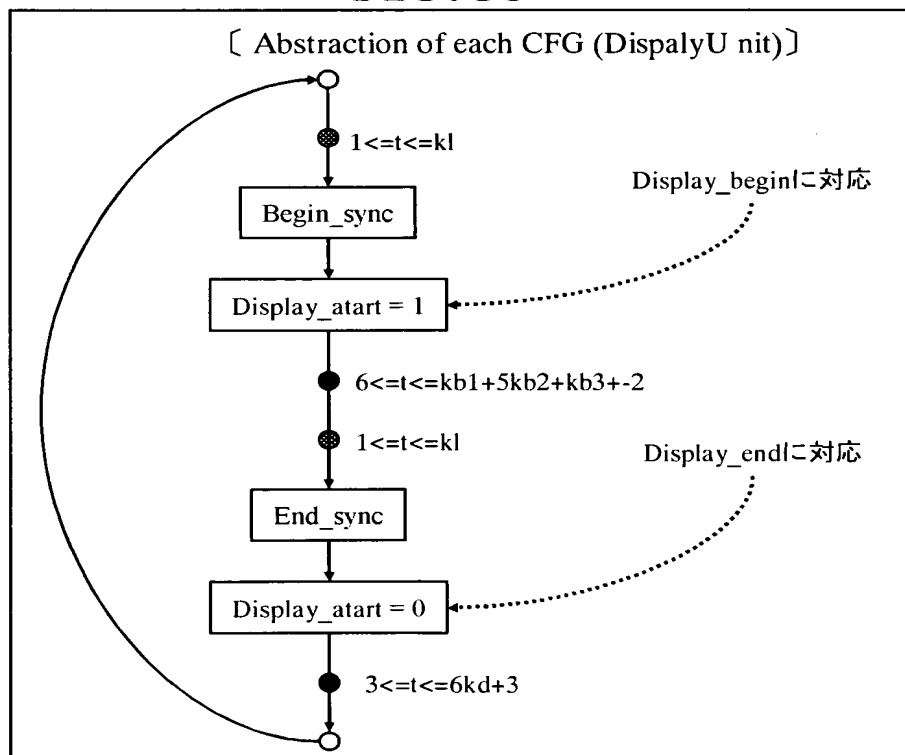
FIG. 55**FIG. 58**

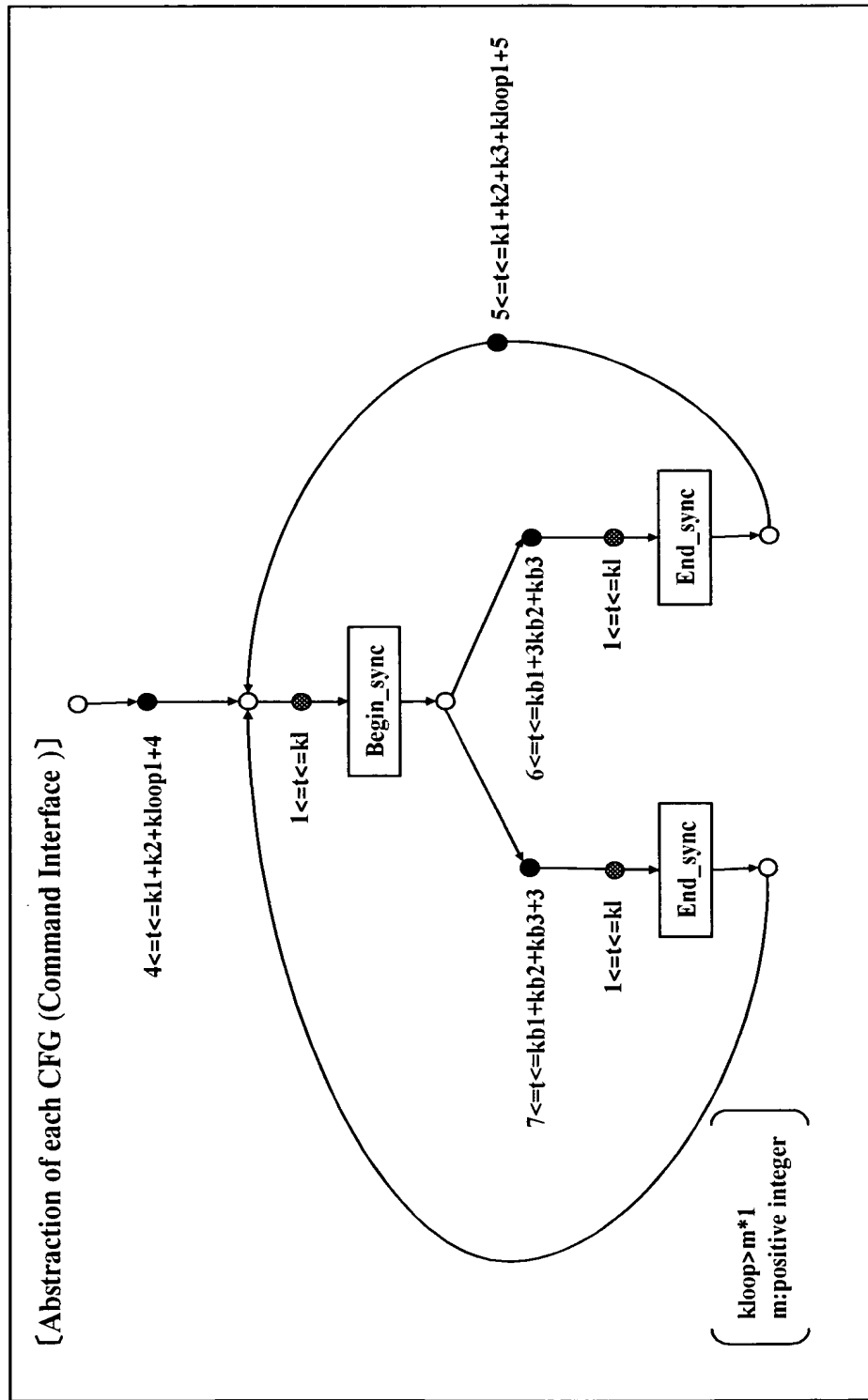
FIG. 56

FIG. 57

[Abstraction of each CFG (Graphics Rendering Unit)]

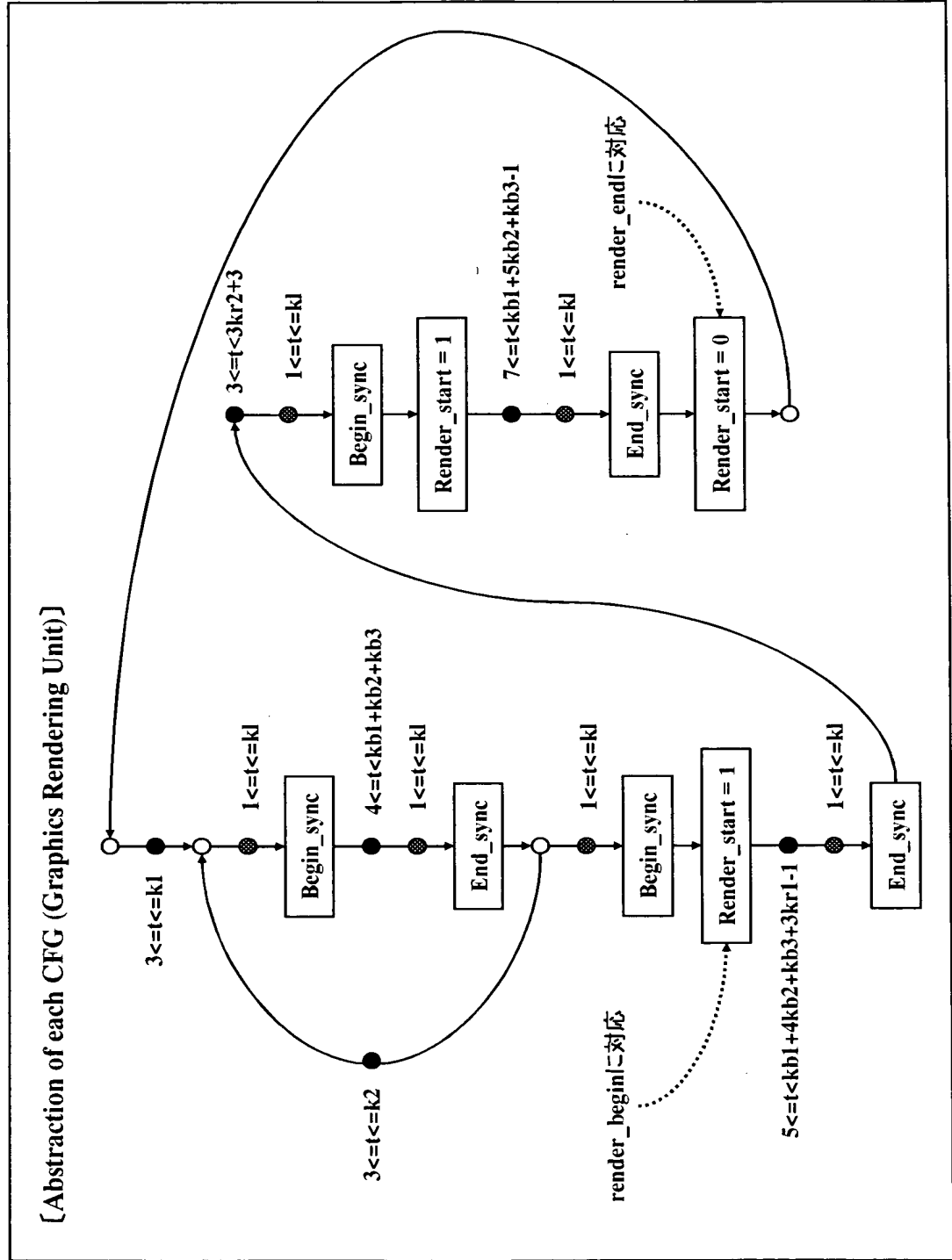


FIG. 59

[C-CFG2C-TNFA (Command Interface)]

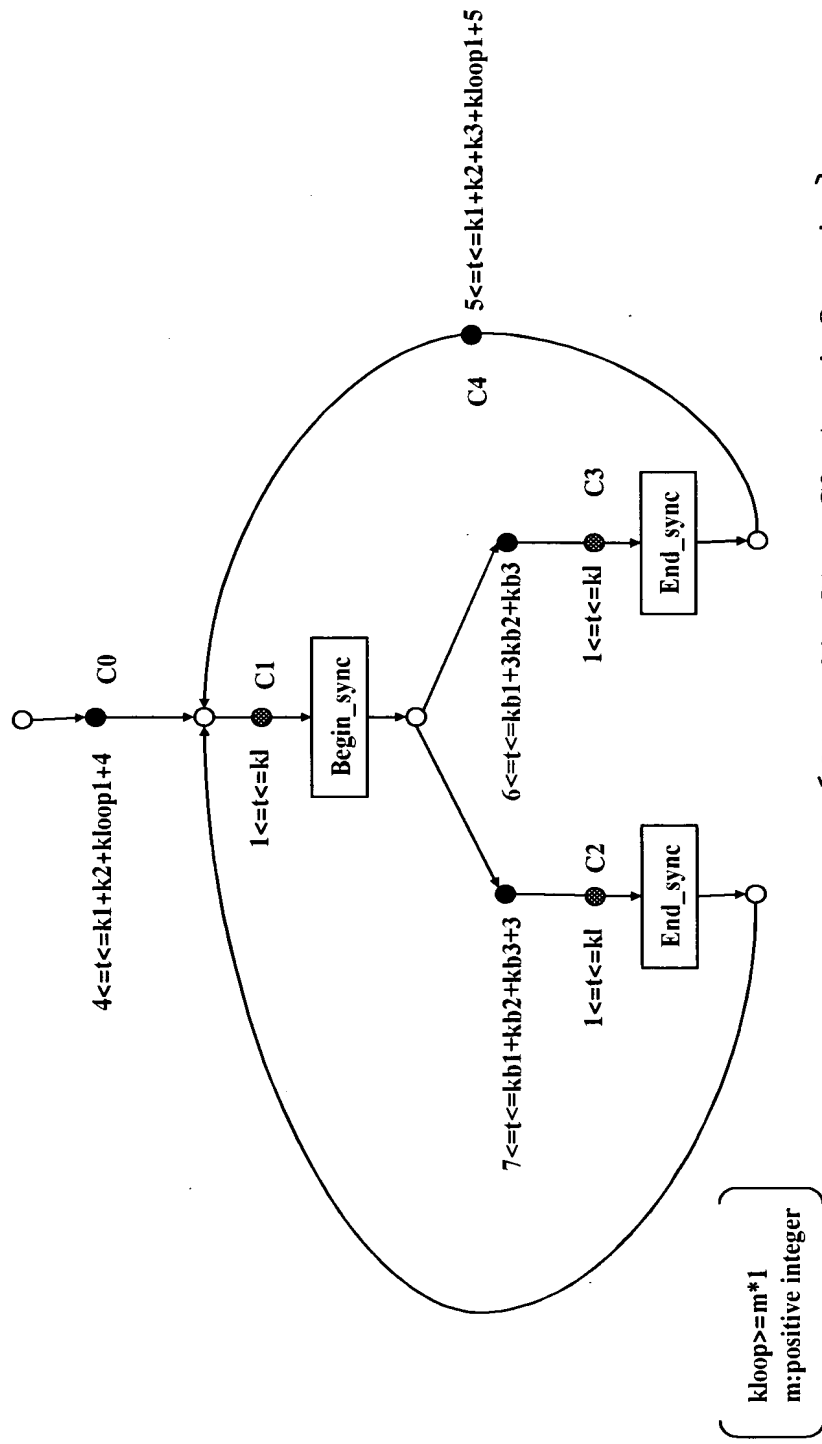


FIG. 60

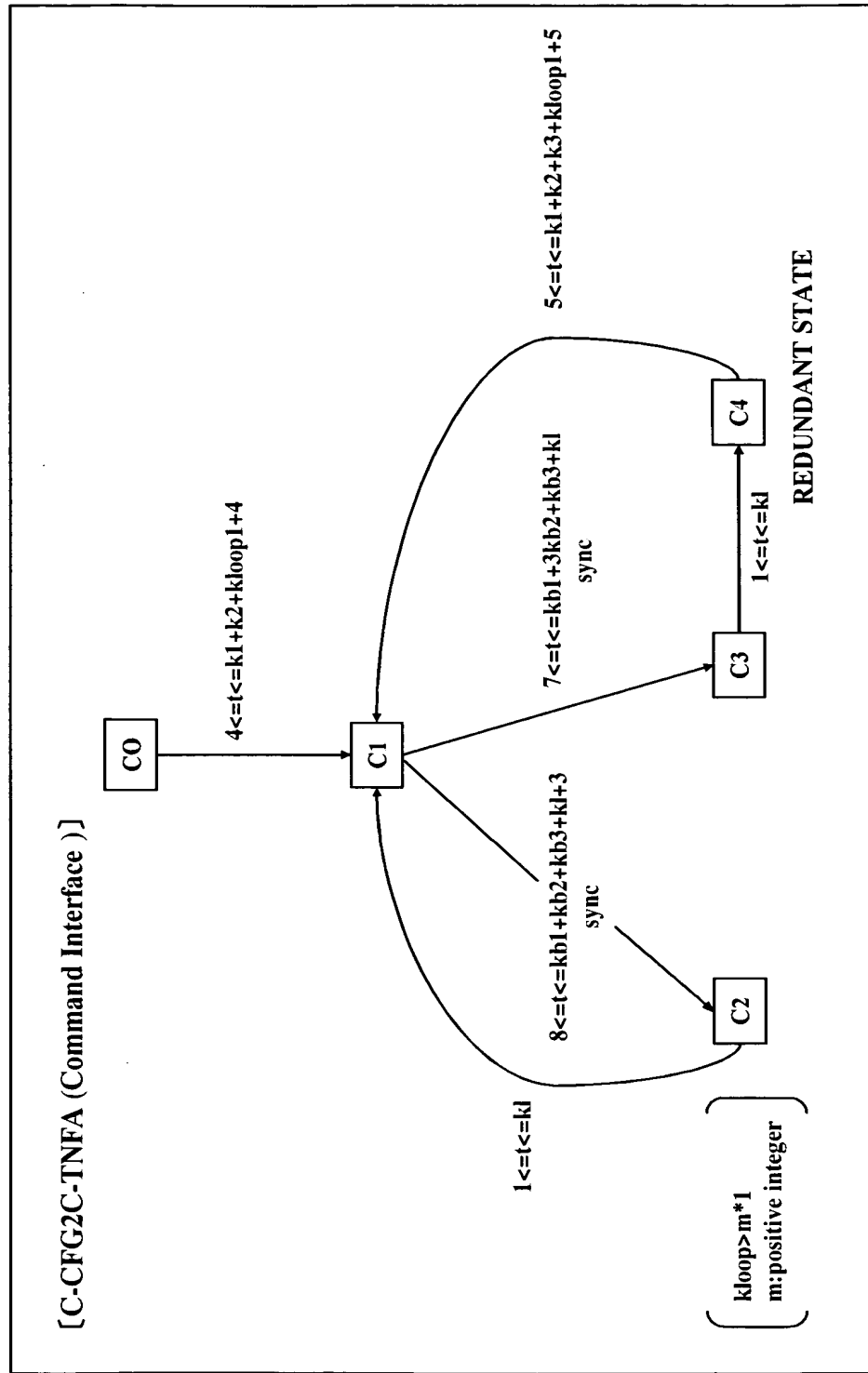


FIG. 61

[C-CFG2C-TNFA (Command Interface)]

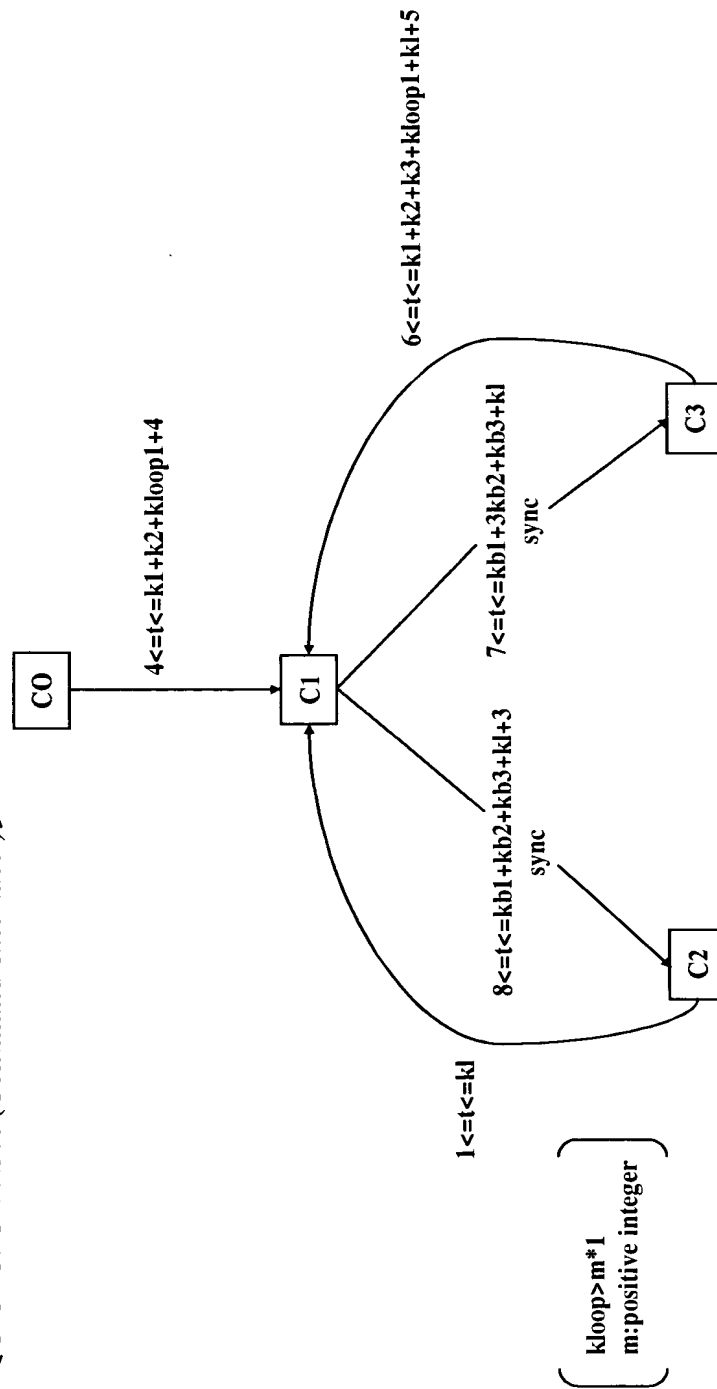


FIG. 62

[C-CFG2C-TNFA (Graphics Rendering Unit)]

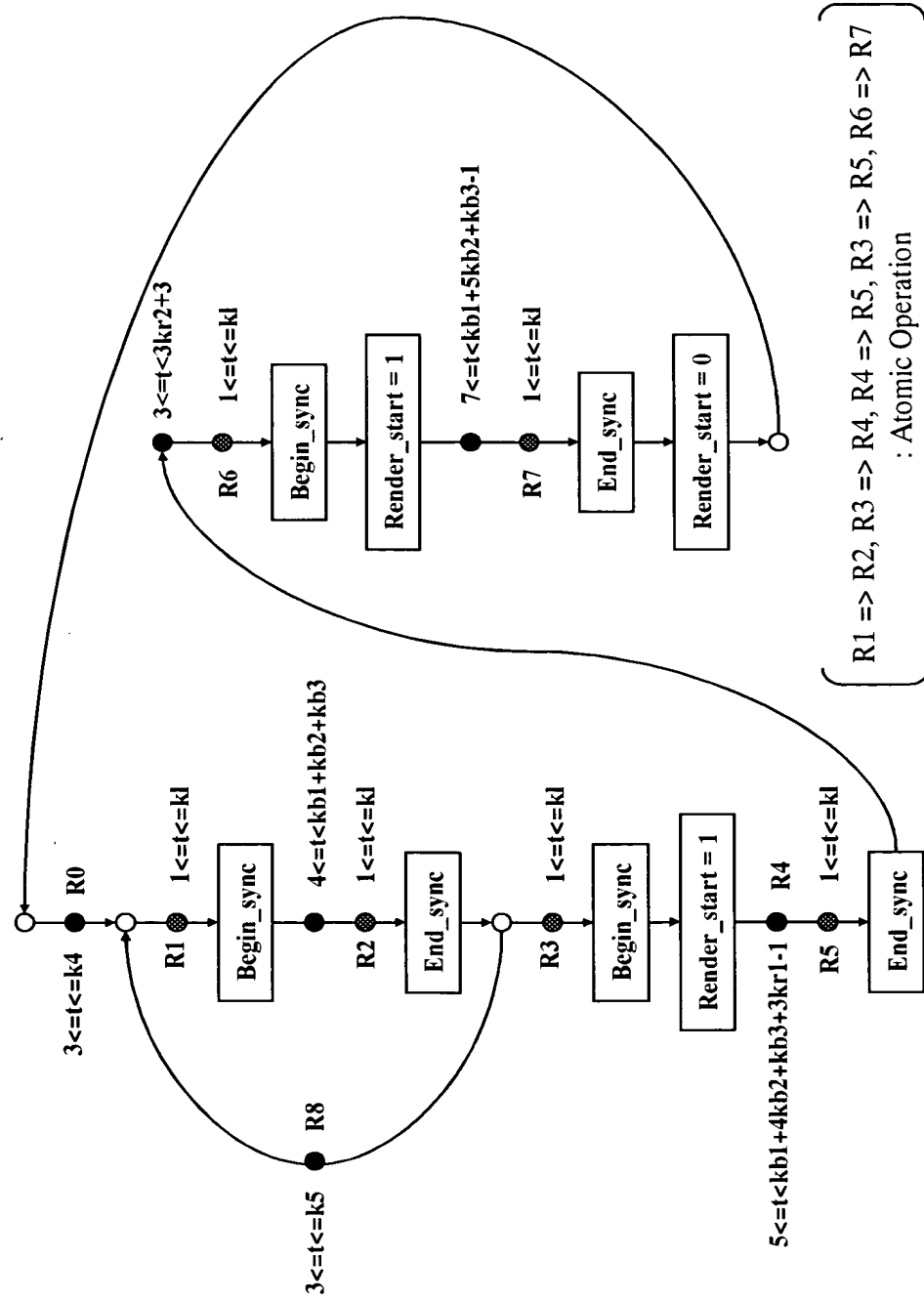


FIG. 63

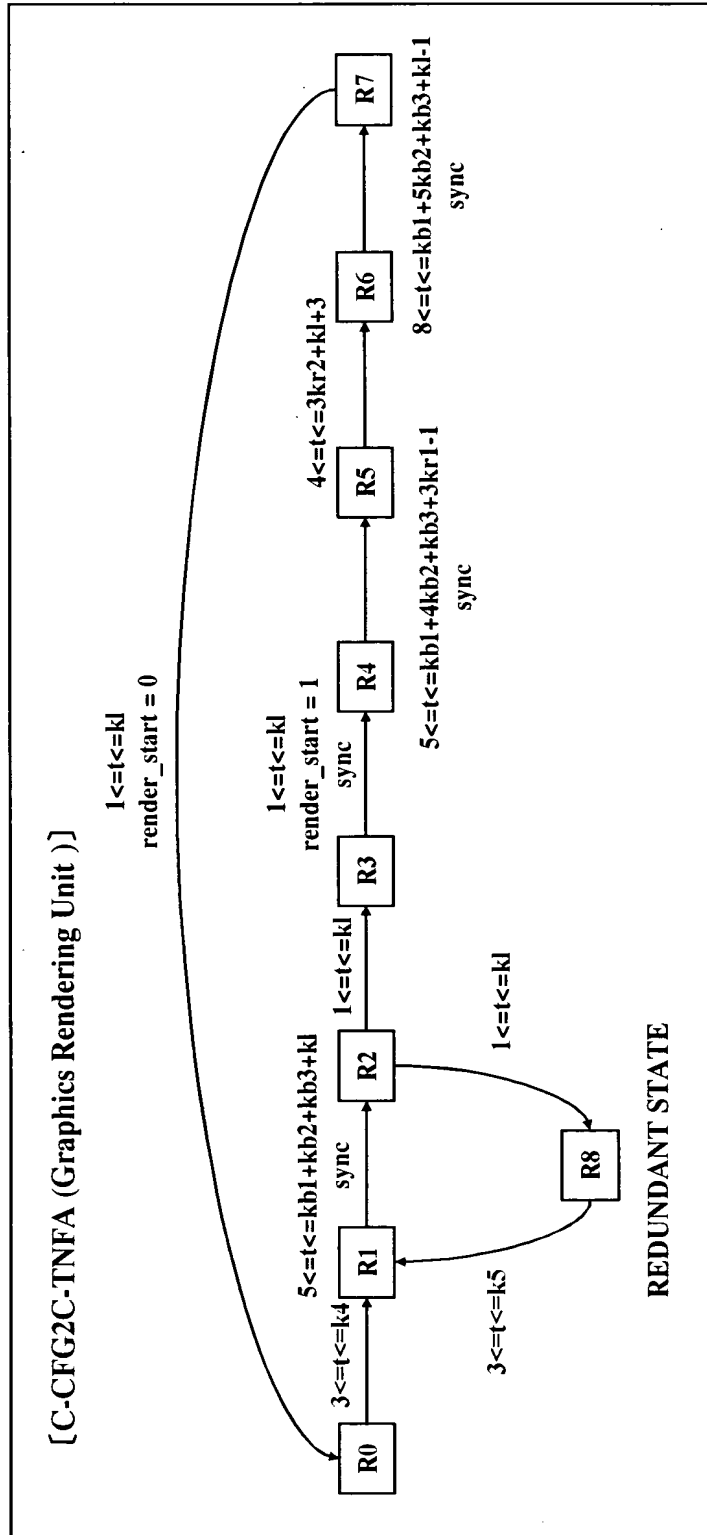


FIG. 64

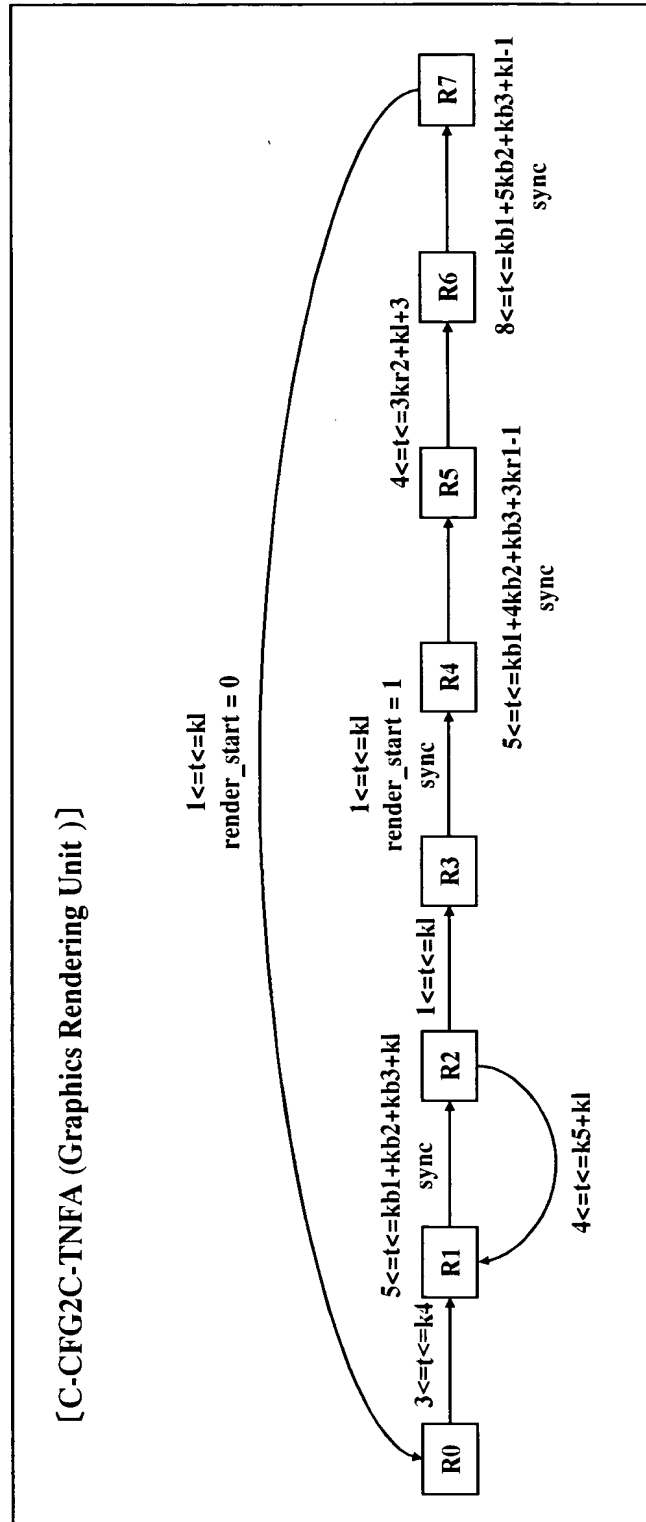


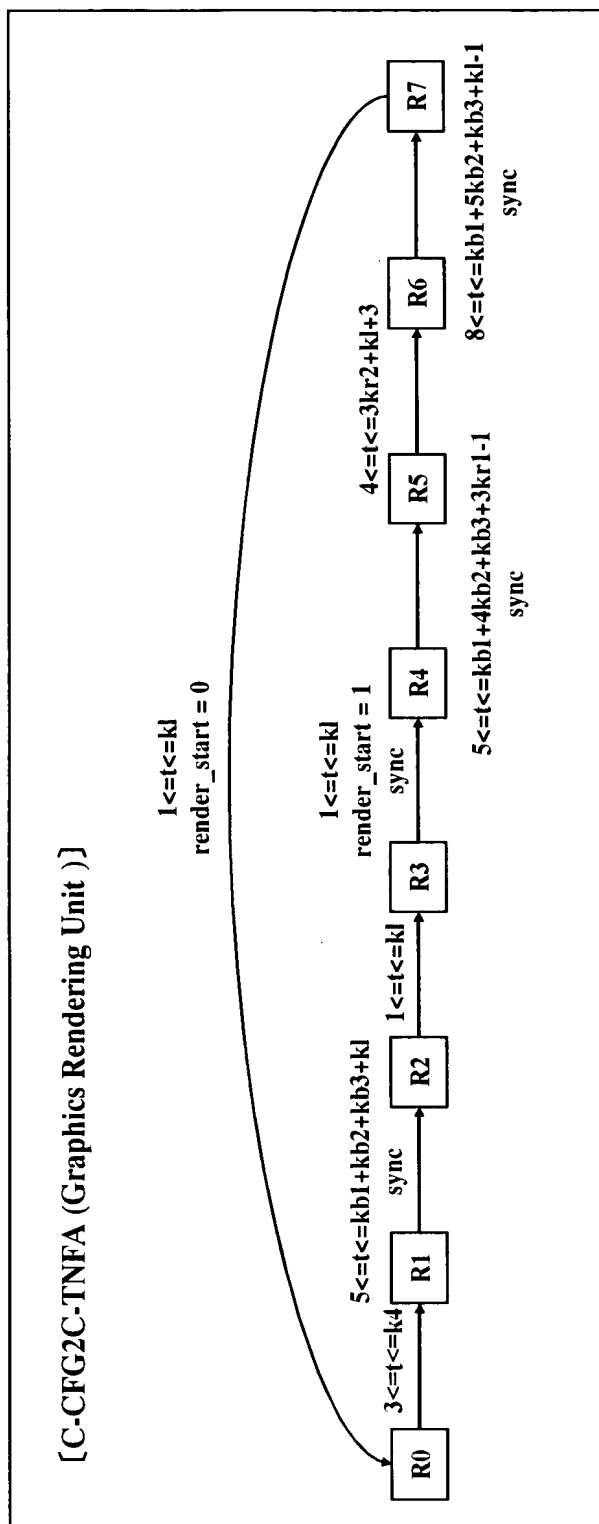
FIG. 65

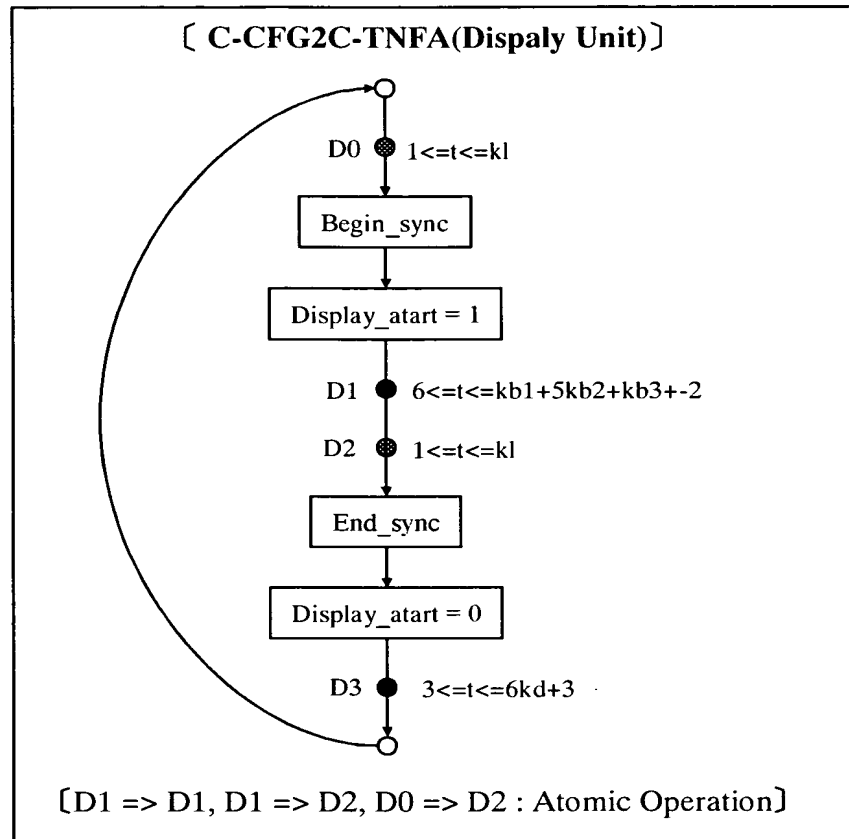
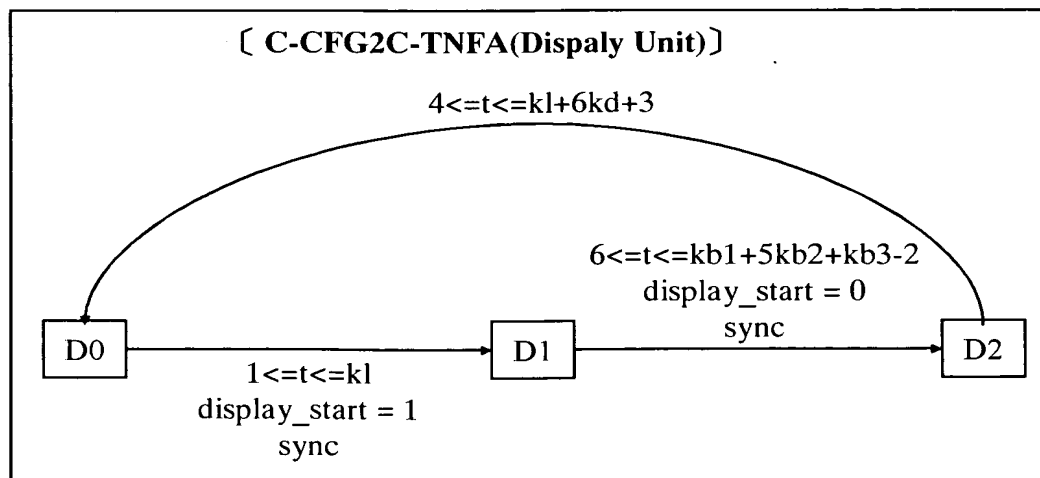
FIG. 66**FIG. 68**

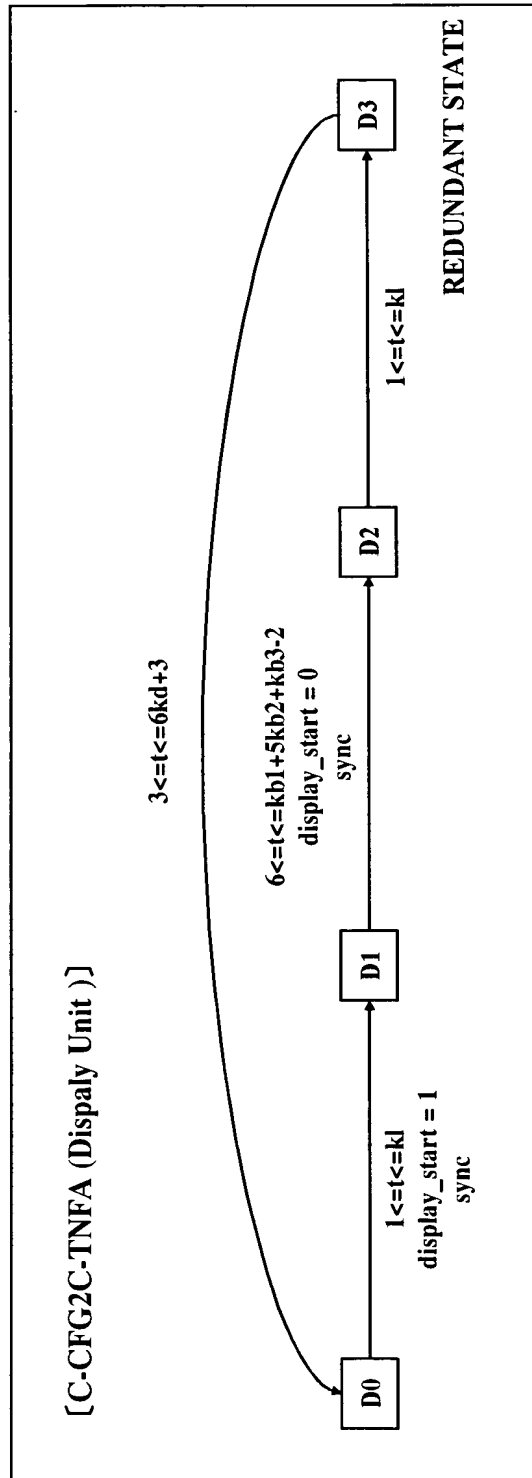
FIG. 67

FIG. 69

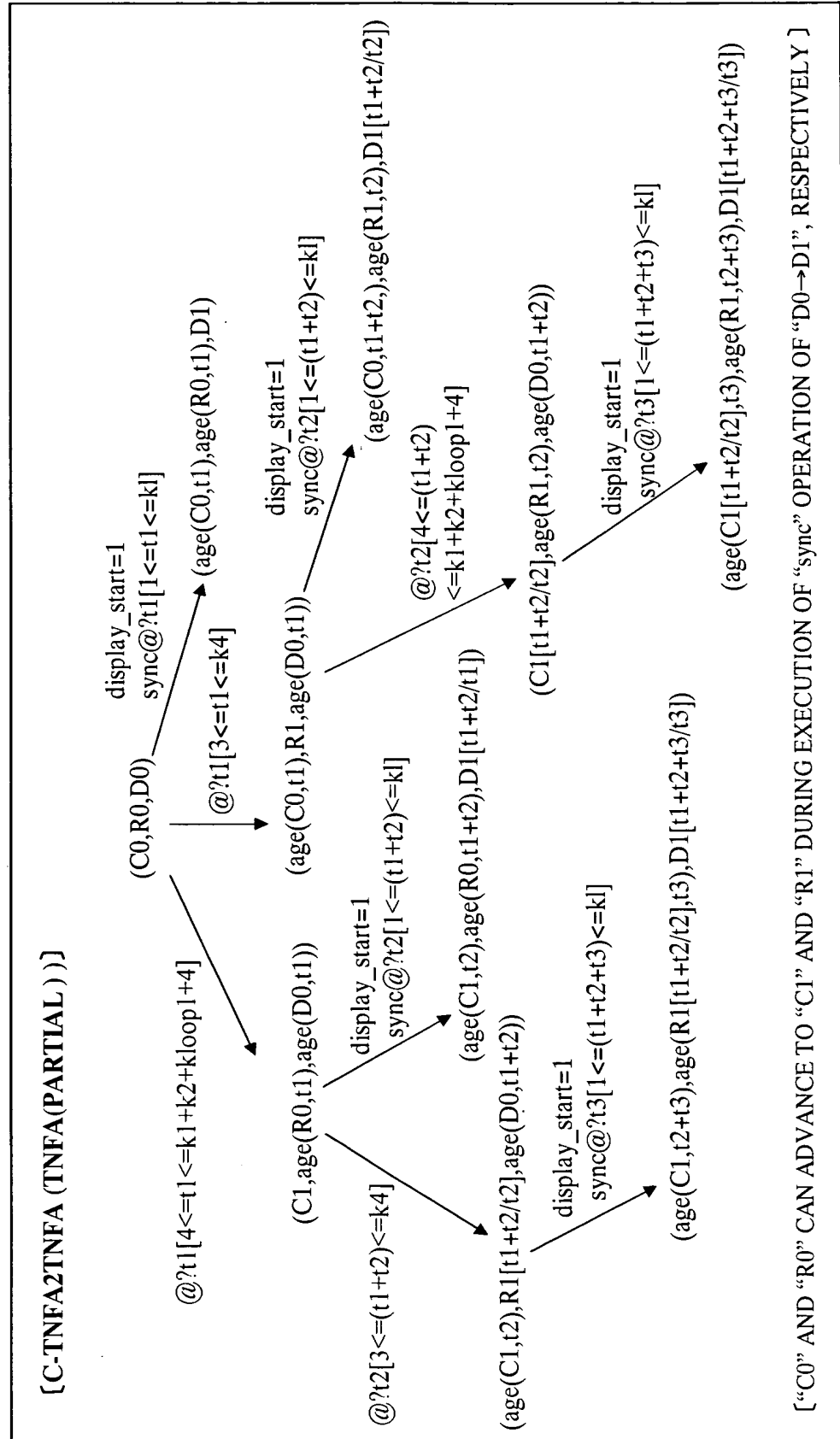


FIG. 70

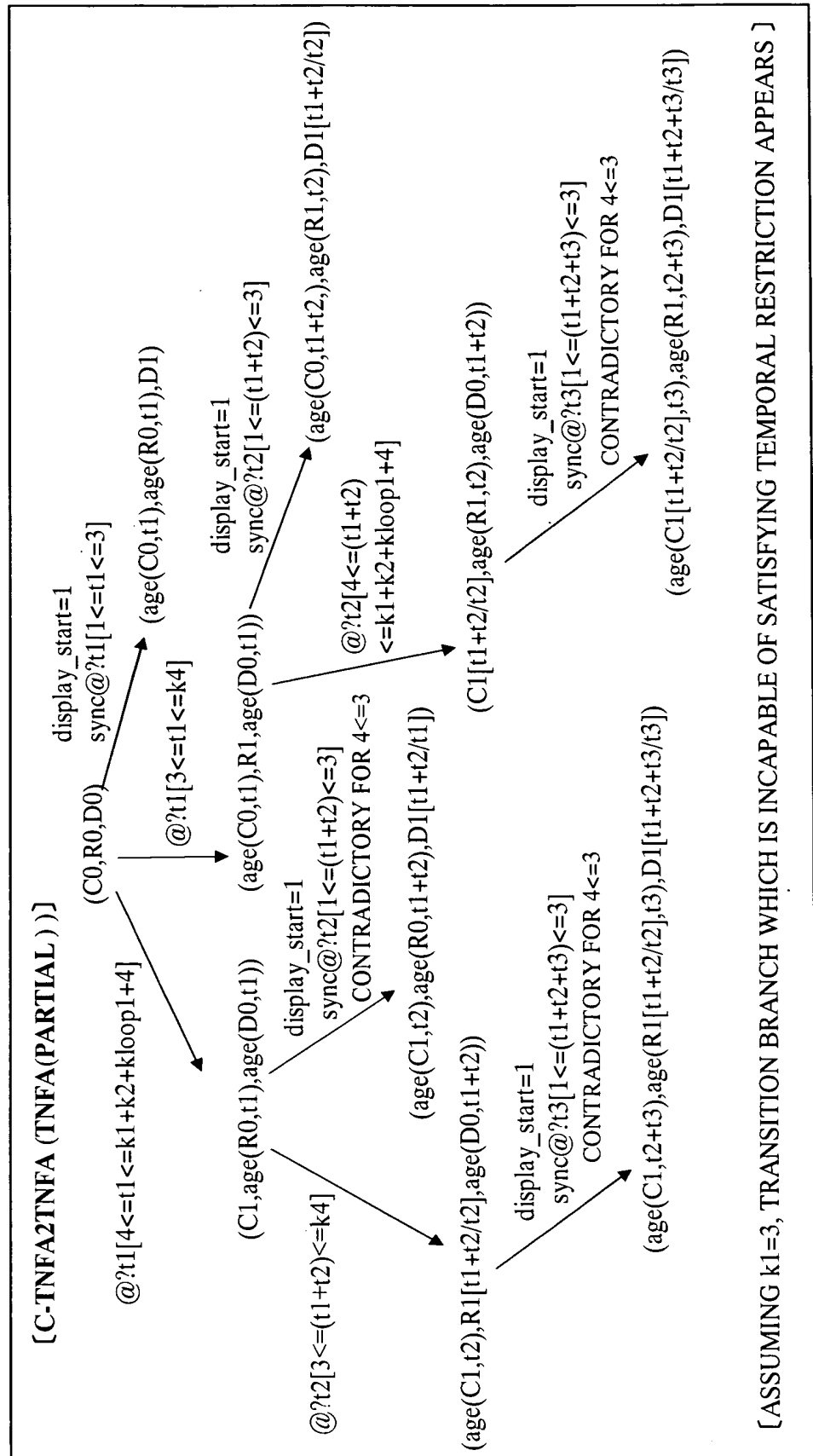


FIG. 71

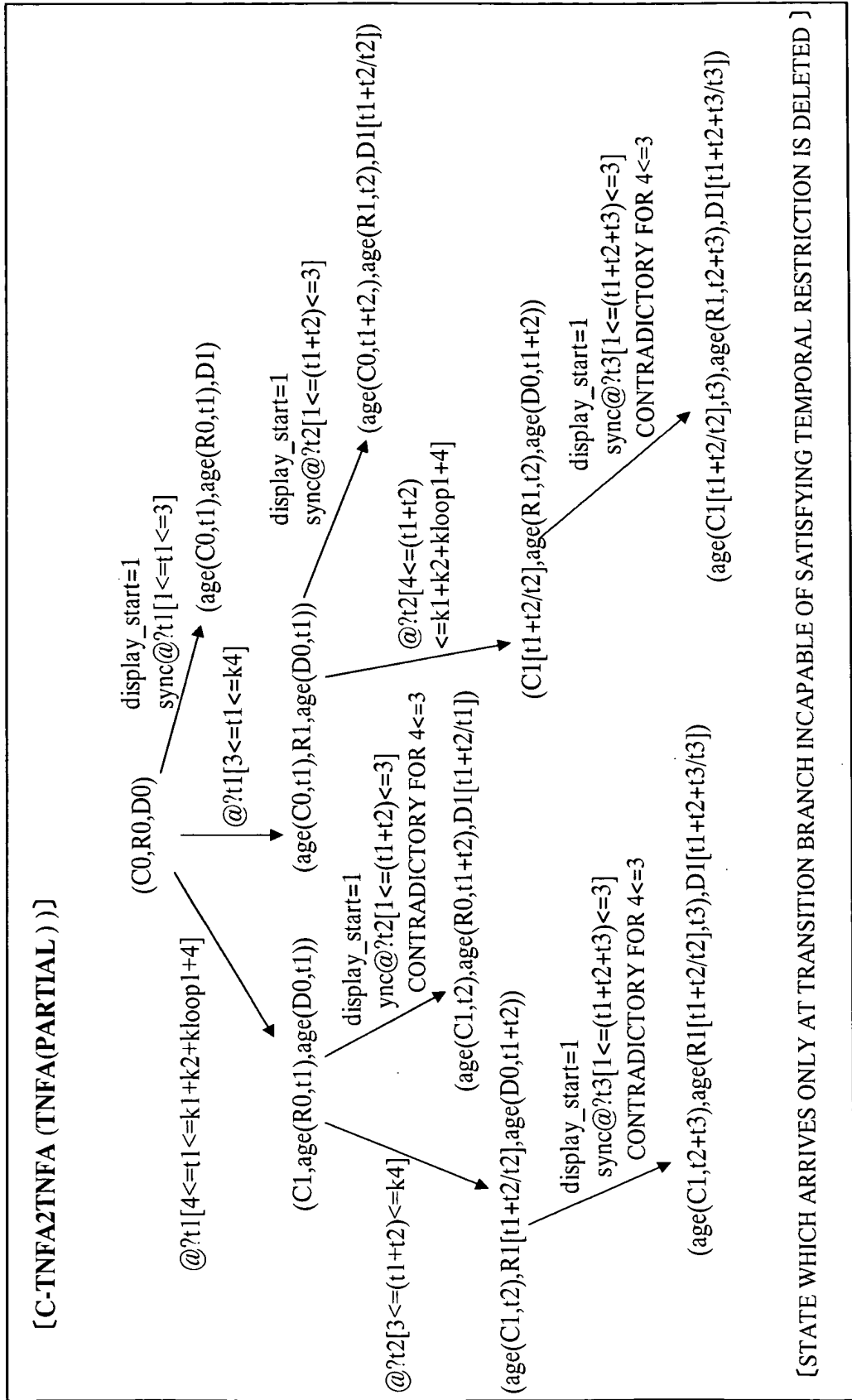
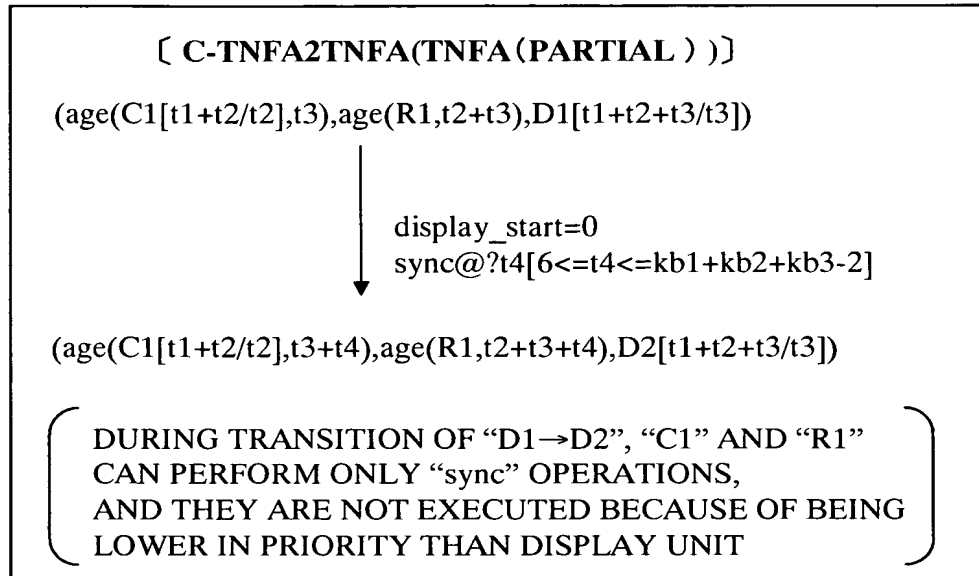


FIG. 72**FIG. 80**

**[PARAMETRIC ANALYSIS RESULT
(EXECUTION RESULT EXAMPLE)]**

Value of objective function: 12

k1	0
k2	0
k3	0
k4	4
k5	0
kd	0
kl	4
kloop1	0
kb1	3
kb2	1
kb3	0
kr1	0
kr2	0

FIG. 73

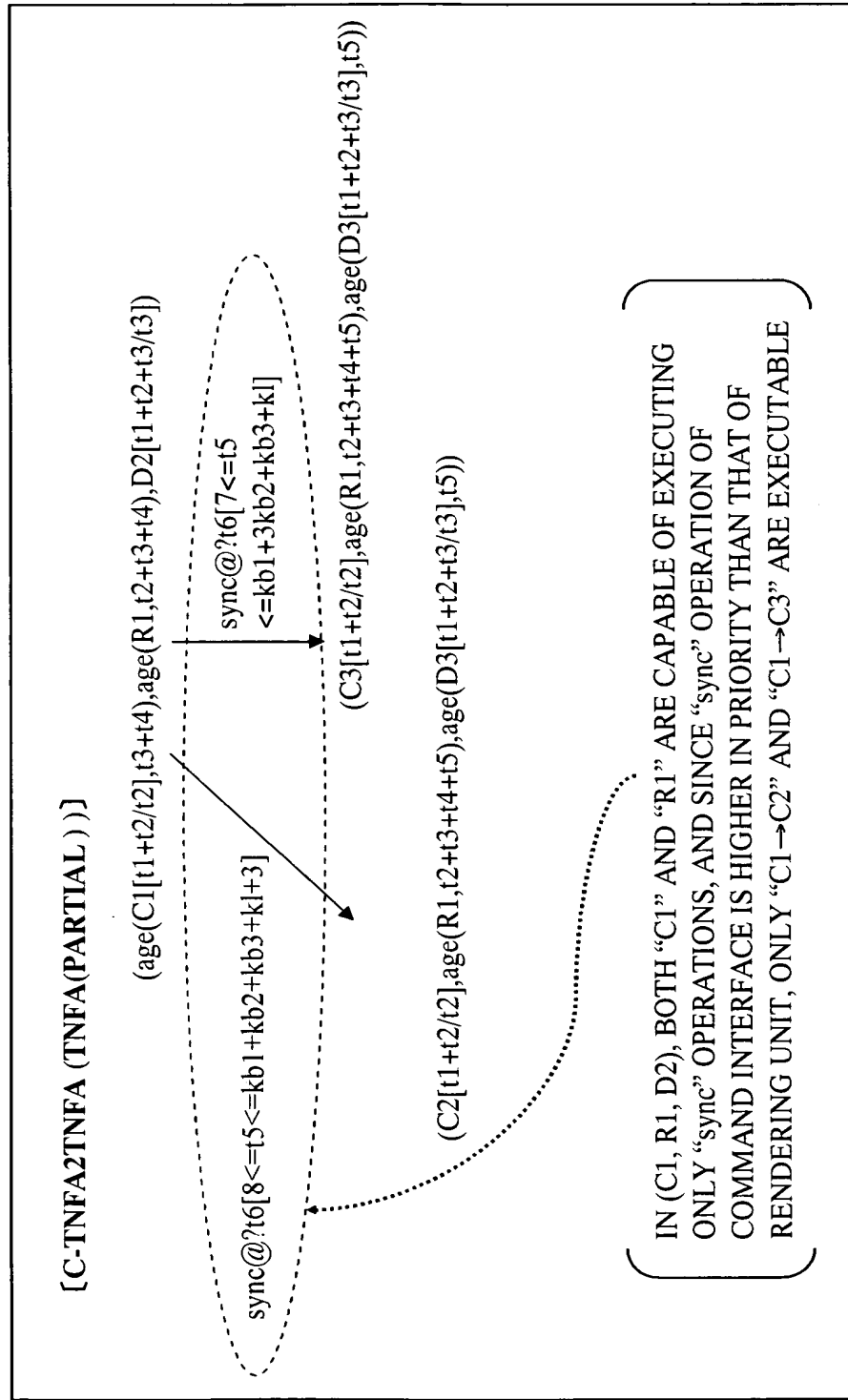


FIG. 74

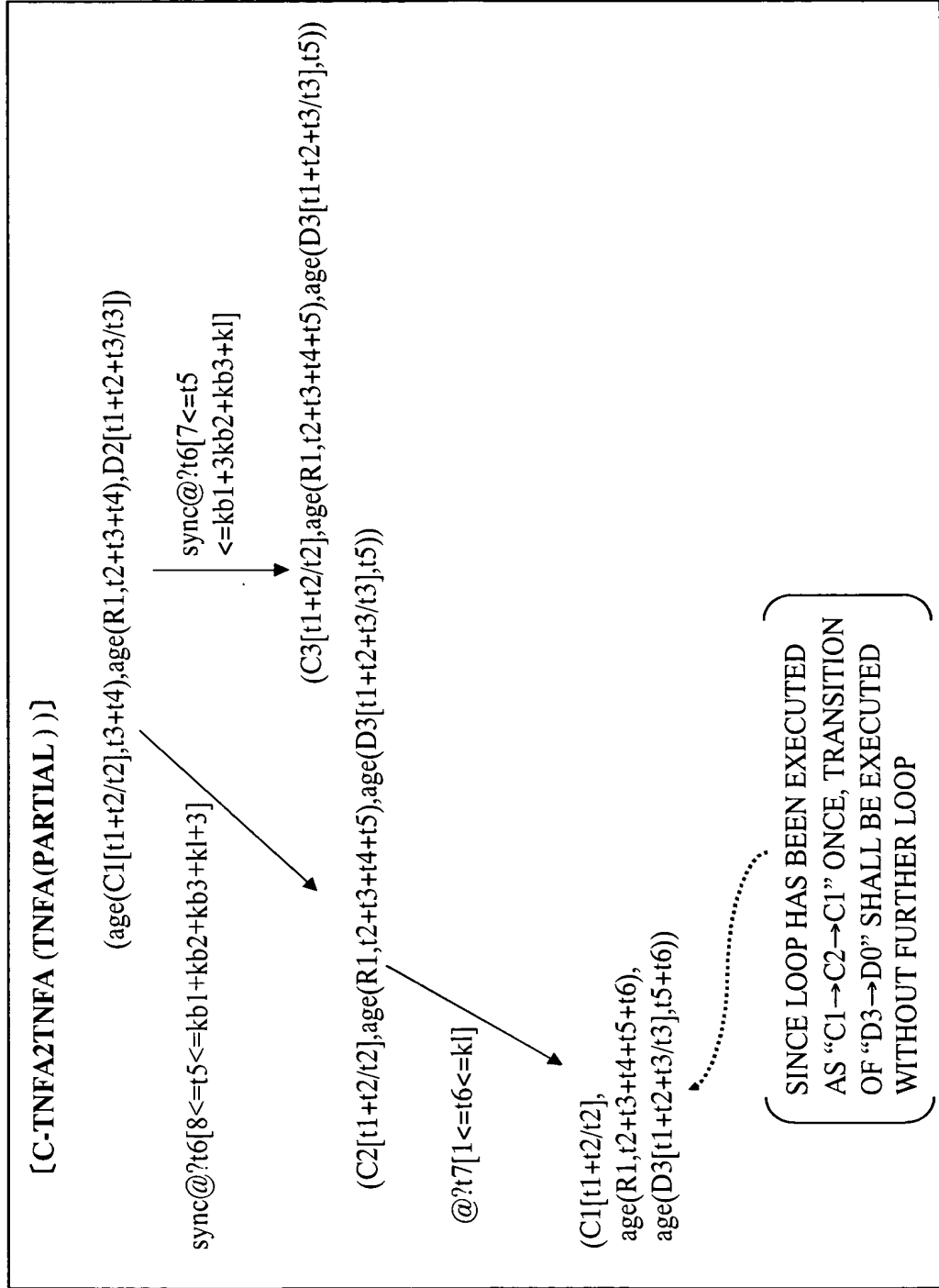


FIG. 75

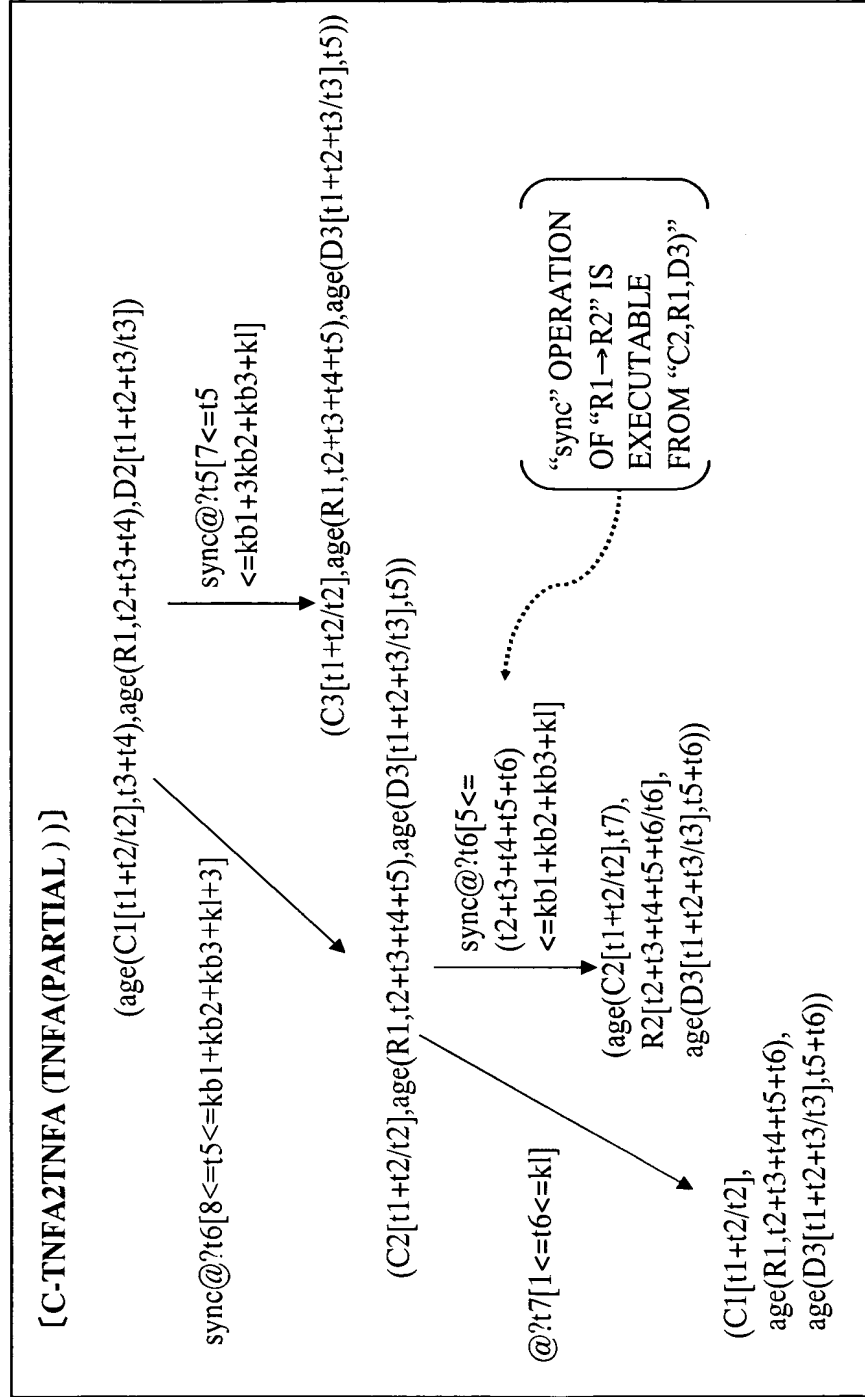


FIG. 76

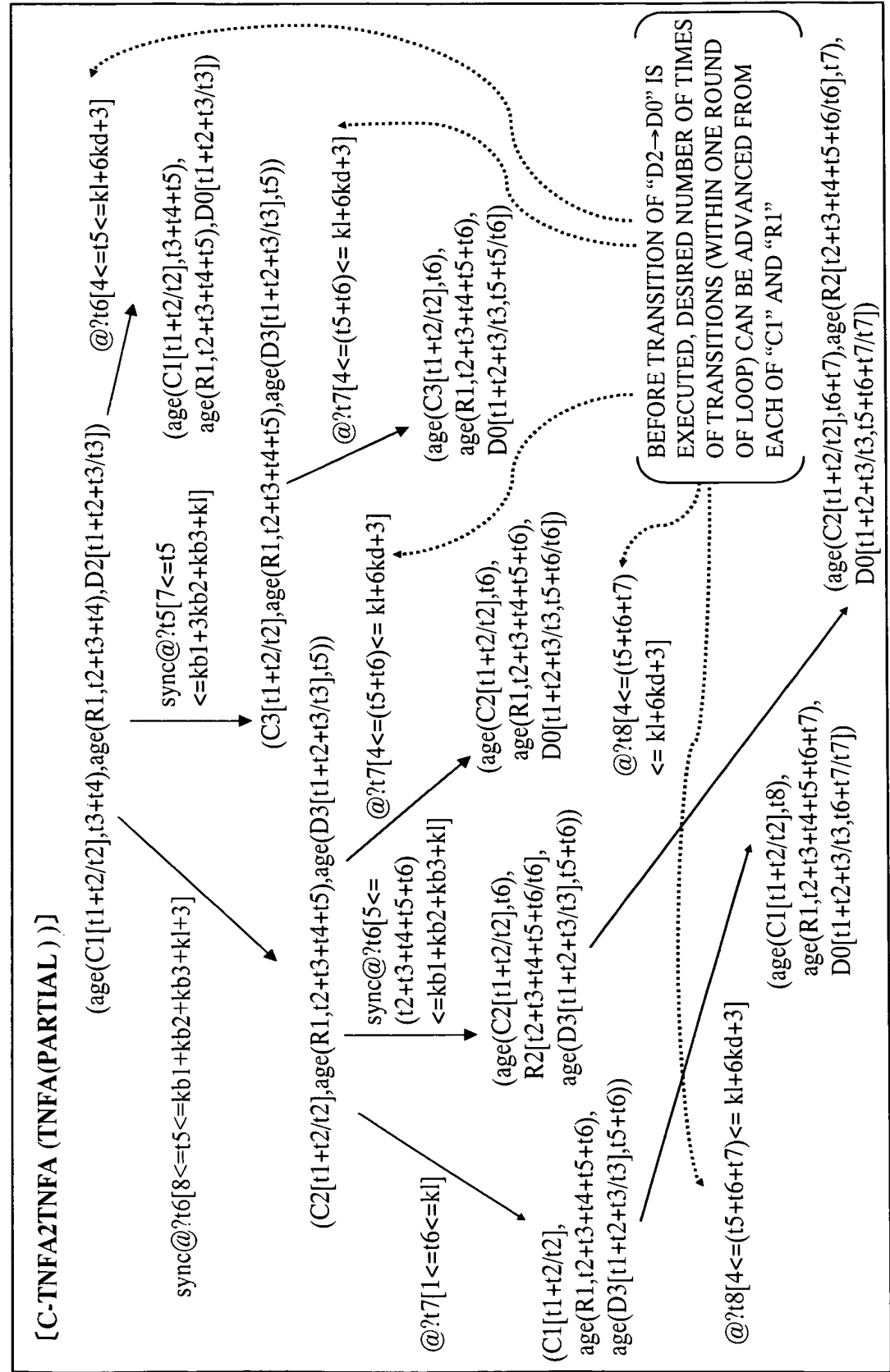


FIG. 77

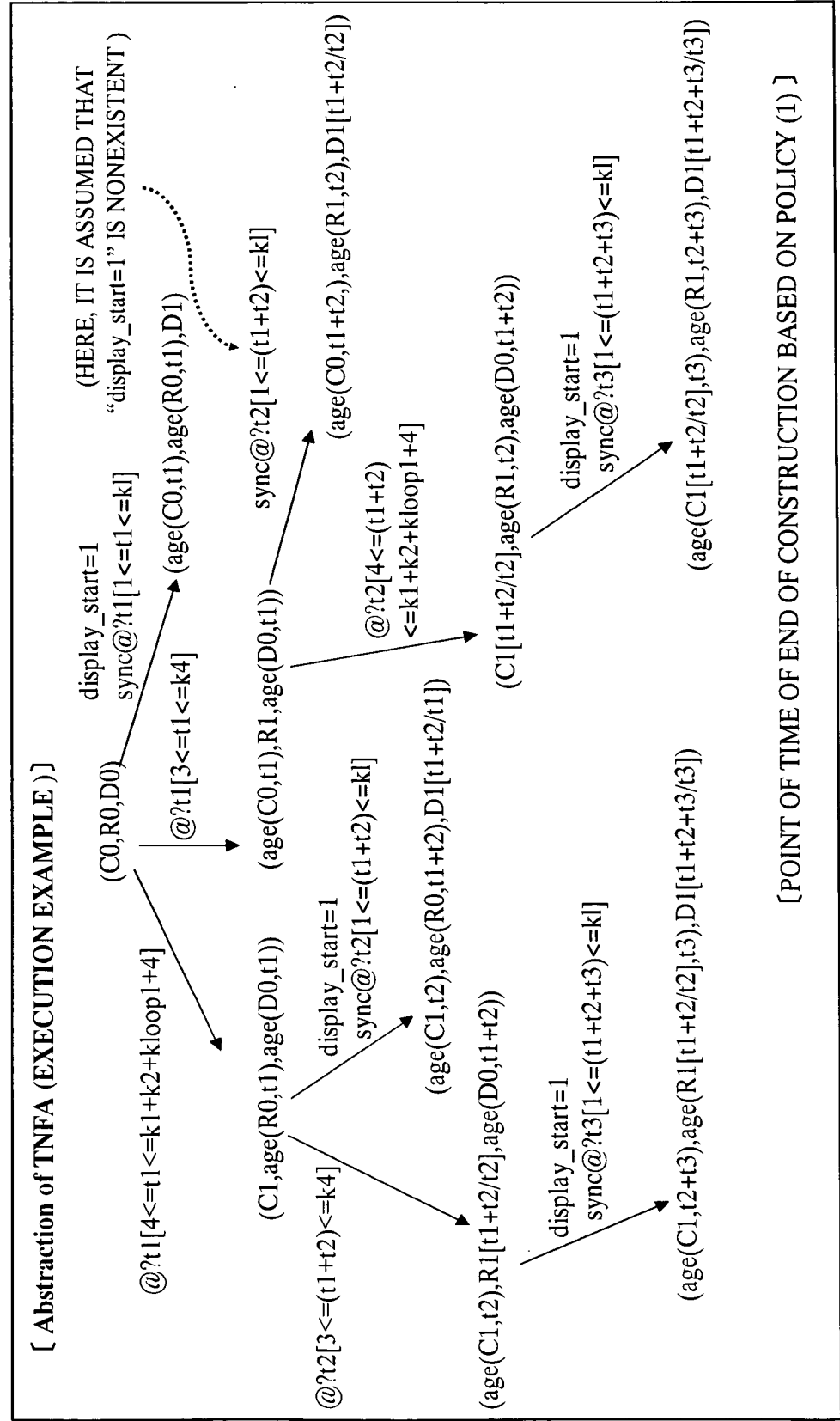


FIG. 78

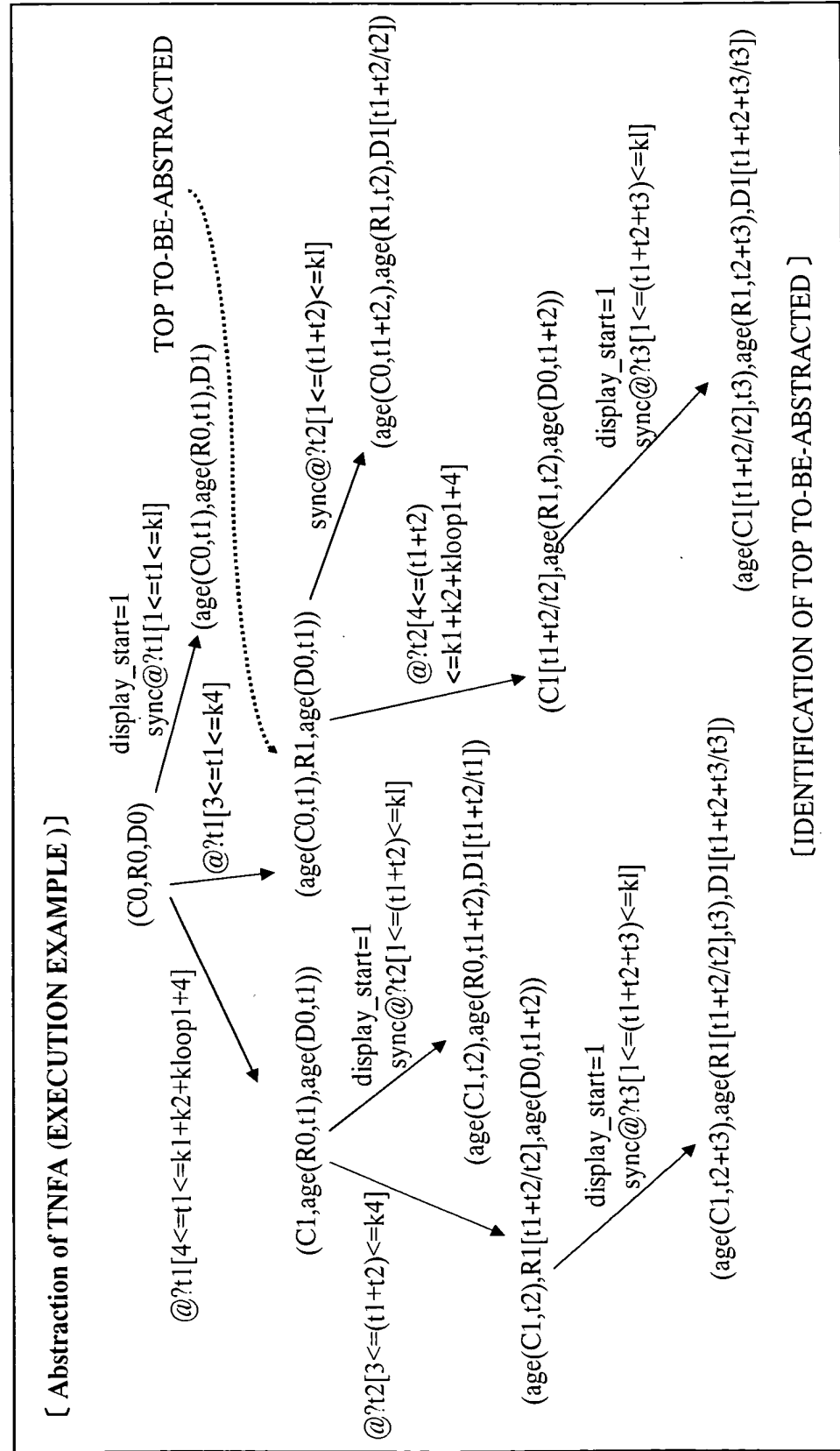


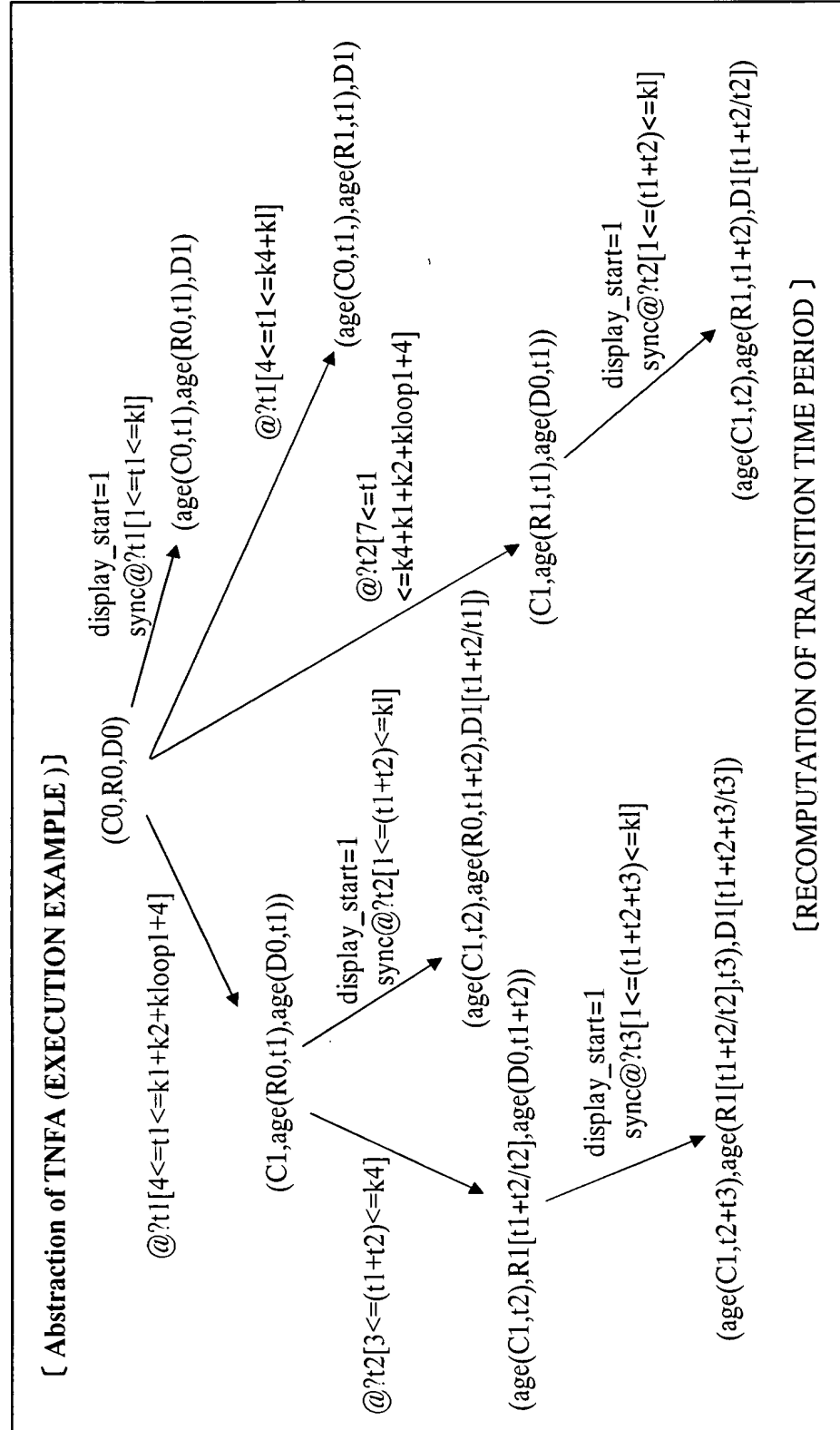
FIG. 79

FIG. 81

**[PARAMETRIC ANALYSIS RESULT
(EXECUTION RESULT EXAMPLE)]**

Value of objective function: 12

k1	0
k2	0
k3	0
k4	4
k5	0
kd	0
kl	4
kloop1	0
kb1	2
kb2	1
kb3	1
kr1	0
kr2	0

FIG. 82

**[PARAMETRIC ANALYSIS RESULT
(EXECUTION RESULT EXAMPLE)]**

Value of objective function: 19

k1	1
k2	1
k3	1
k4	4
k5	1
kd	1
kl	4
kloop1	1
kb1	2
kb2	1
kb3	1
kr1	0
kr2	1

FIG. 83

[HARDWARE SYNTHESIS (ALLOTMENT OF EXECUTION CYCLES TO BASIC BLOCK)]

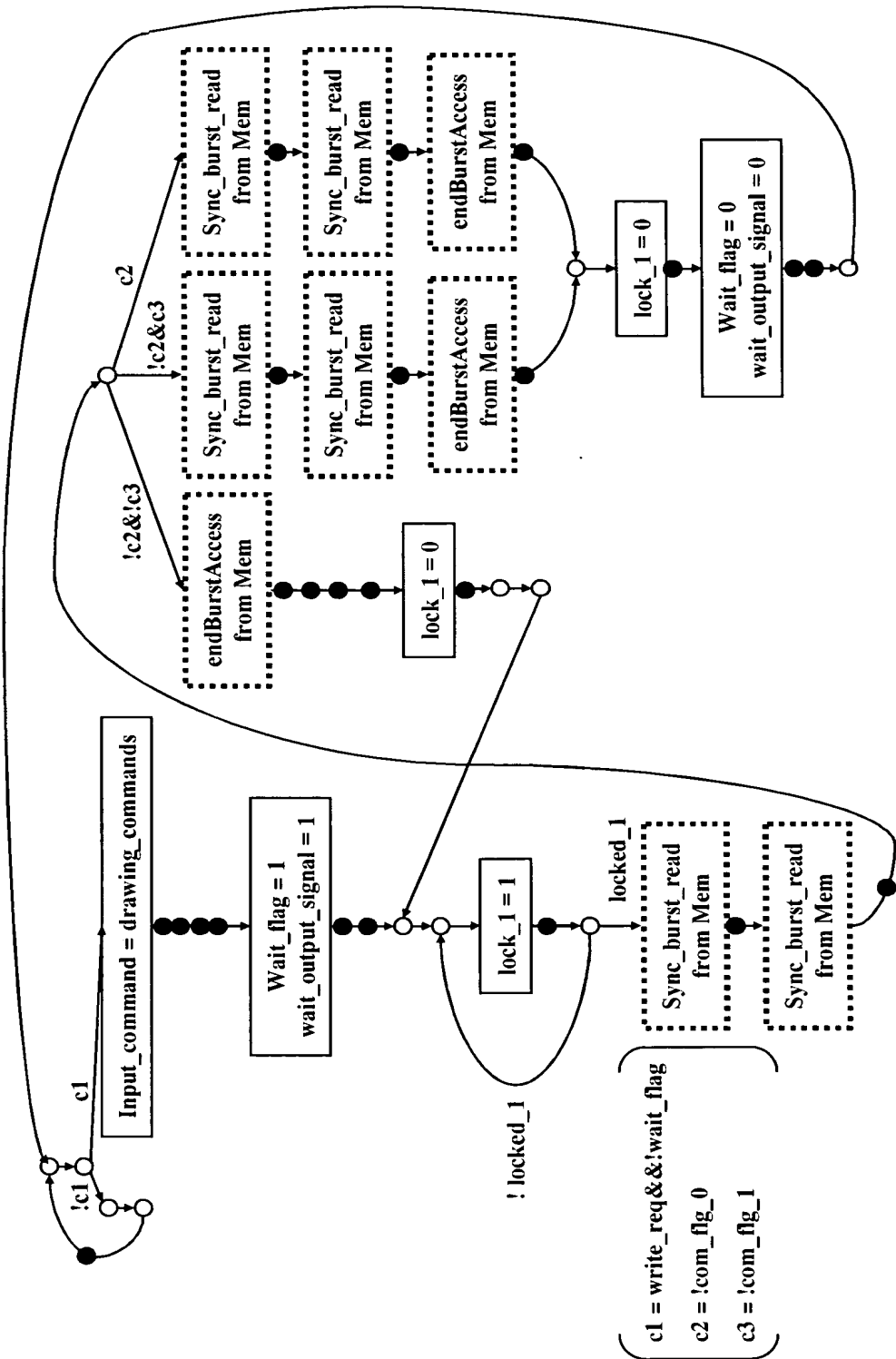


FIG. 84

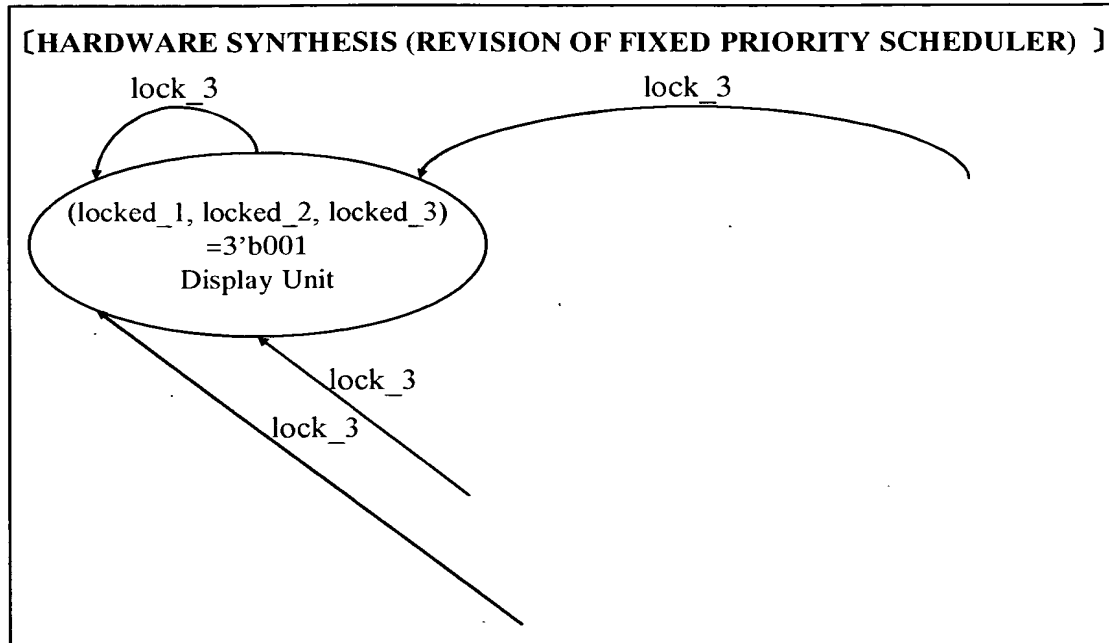


FIG. 90

[HARDWARE SYNTHESIS (SHARED REGISTER)]

```

void register_in() {
  if (AD_BUS[3:0] == 4'b0000) {
    mem_con_reg.current_value[0] = $ D_BUS;
  } else if (AD_BUS[3:0] == 4'b0001) {
    mem_con_reg.current_value[1] = $ D_BUS;
  } else if (AD_BUS[3:0] == 4'b0010) {
    mem_con_reg.current_value[2] = $ D_BUS;
  } else if (AD_BUS[3:0] == 4'b0011) {
    mem_con_reg.current_value[3] = $ D_BUS;
  } else if (AD_BUS[3:0] == 4'b0100) {
    mem_con_reg.current_value[4] = $ D_BUS;
  } else if (AD_BUS[3:0] == 4'b0101) {
    mem_con_reg.current_value[5] = $ D_BUS;
  } else if (AD_BUS[3:0] == 4'b0110) {
    mem_con_reg.current_value[6] = $ D_BUS;
  } else if (AD_BUS[3:0] == 4'b0111) {
    mem_con_reg.current_value[7] = $ D_BUS;
  } else if (AD_BUS[3:0] == 4'b1000) {
    mem_con_reg.current_value[8] = $ D_BUS;
  } else if (AD_BUS[3:0] == 4'b1001) {
    mem_con_reg.current_value[9] = $ D_BUS;
  }
}

```

FIG. 85

[HARDWARE SYNTHESIS (REVISION OF FIXED PRIORITY SCHEDULER)]

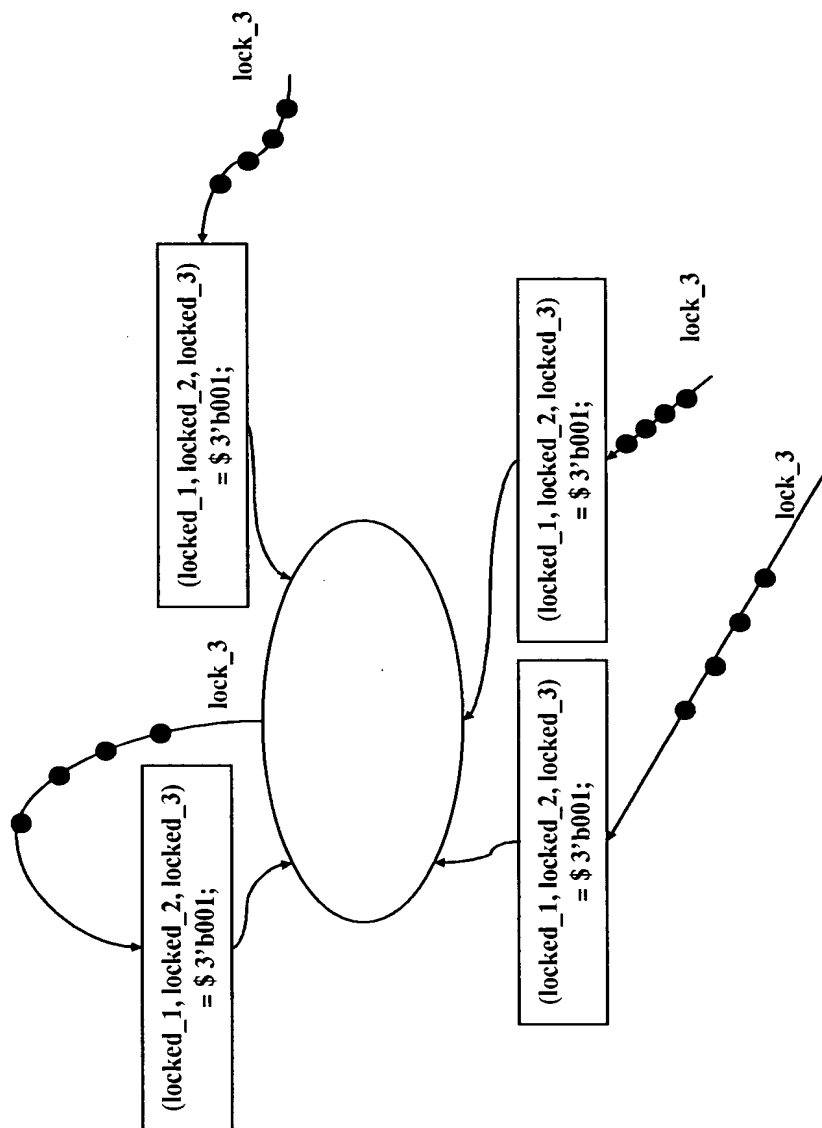


FIG. 86

[HARDWARE SYNTHESIS (TRANSFORMATION OF CFG)]

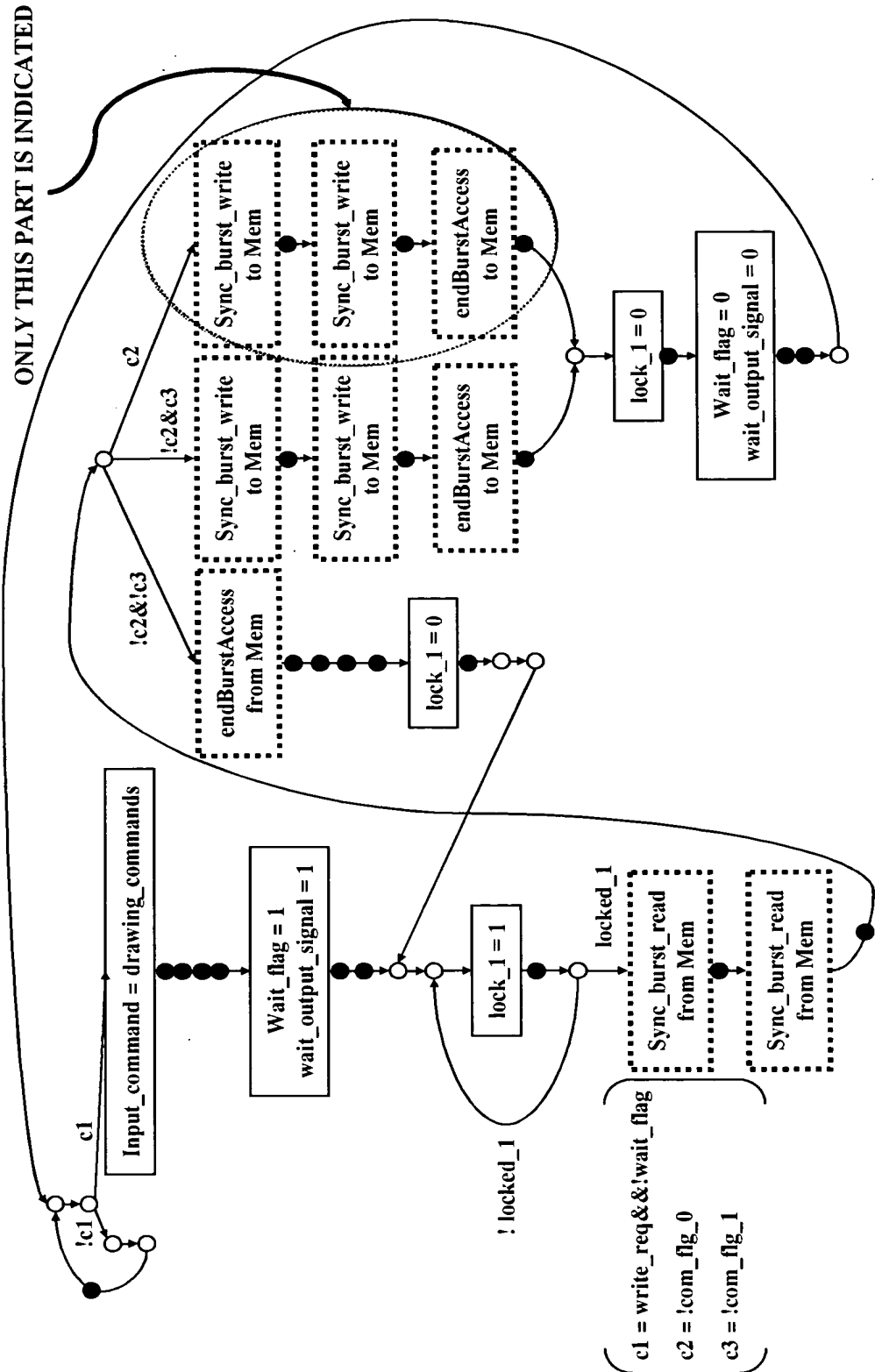


FIG. 87

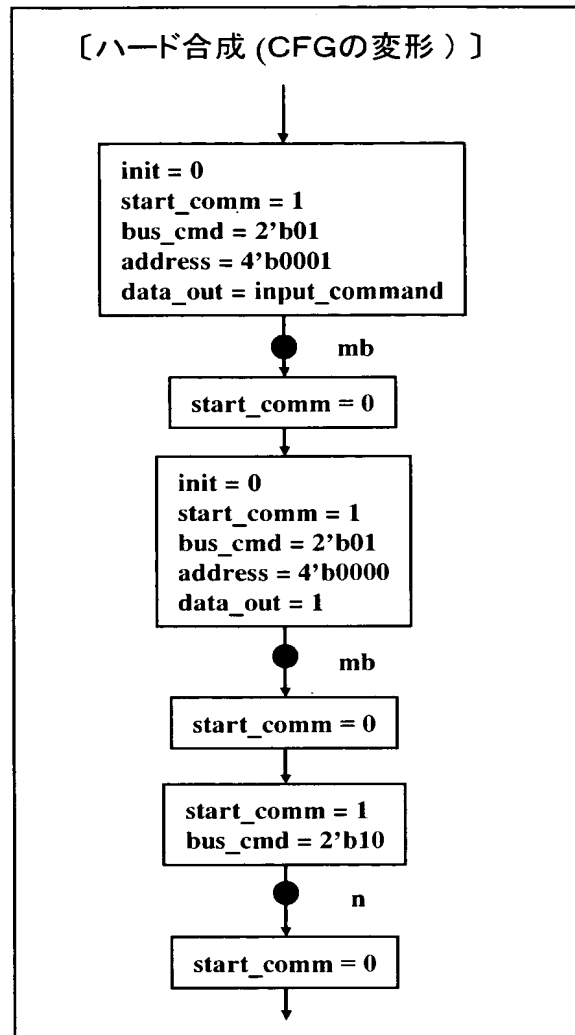


FIG. 89**[HARDWARE SYNTHESIS (SHARED REGISTER)]**

```

while(1) {
  D_BUS = 32'bz;
  if (locked_1 || locked_2 || locked_3) {
    if (AD_BUS[5:4] == 2'b00) {
      if (init) {
        L1:
          if (data_out_en_1 || data_out_en_2 || data_out_en_3) {
            $
              register_in();
            } else {
              $ goto L1;
            }
          } else {
            L2:
              if (data_out_en_1 || data_out_en_2 || data_out_en_3) {
                register_in();
              } else {
                $ goto L2;
              }
            }
          }
        }
      }
    }
  }
}

```

```

} else if (AD_BUS == 2'b01) {
  if (init) {
    L3:
      if (data_in_en_1 || data_in_en_2 || data_in_en_3) {
        $
          register_out();
        } else {
          $ goto L3;
        }
      } else {
        L4:
          if (data_in_en_1 || data_in_en_2 || data_in_en_3) {
            register_out();
          } else {
            $ goto L4;
          }
        }
      }
    }
  }
}

```

FIG. 91

〔ハード合成 (共有レジスタ) 〕

```
void register_out() {  
    if (AD_BUS[3:0] == 4'b0000) {  
        D_BUS = mem_con_reg.current_value[0];  
    } else if (AD_BUS[3:0] == 4'b0001) {  
        D_BUS = mem_con_reg.current_value[1];  
    } else if (AD_BUS[3:0] == 4'b0010) {  
        D_BUS = mem_con_reg.current_value[2];  
    } else if (AD_BUS[3:0] == 4'b0011) {  
        D_BUS = mem_con_reg.current_value[3];  
    } else if (AD_BUS[3:0] == 4'b0100) {  
        D_BUS = mem_con_reg.current_value[4];  
    } else if (AD_BUS[3:0] == 4'b0101) {  
        D_BUS = mem_con_reg.current_value[5];  
    } else if (AD_BUS[3:0] == 4'b0110) {  
        D_BUS = mem_con_reg.current_value[6];  
    } else if (AD_BUS[3:0] == 4'b0111) {  
        D_BUS = mem_con_reg.current_value[7];  
    } else if (AD_BUS[3:0] == 4'b1000) {  
        D_BUS = mem_con_reg.current_value[8];  
    } else if (AD_BUS[3:0] == 4'b1001) {  
        D_BUS = mem_con_reg.current_value[9];  
    }  
}
```